O'REILLY®

TURING 图灵程序设计丛书

Java性能优化实践

JVM调优策略、工具与技巧

Optimizing Java

理论与方法相结合，全面阐释JVM领域新知识和一线生产调优经验



[英] 本杰明·J. 埃文斯
詹姆斯·高夫
克里斯·纽兰 著
曾波 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

曾波

互联网架构师，拥有13年互联网从业经验，曾任职于微软、什么值得买、京东金融、鹏博士电信传媒集团，曾参与多家公司的技术团队从建立到成熟的过程，主持实施了重大技术决策和技术落地，在大规模复杂系统架构和技术管理方面拥有丰富的实践经验。

数字版权声明

TURING 图灵程序设计丛书

Java性能优化实践：JVM调优策略、工具与技巧

Optimizing Java: Practical Techniques for Improving JVM Application Performance

[英] 本杰明·J. 埃文斯　詹姆斯·高夫　克里斯·纽兰 著

曾波 译

Beijing • Boston • Farnham • Sebastopol • Tokyo　O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（CIP）数据

Java性能优化实践 : JVM调优策略、工具与技巧 / (英) 本杰明· J.埃文斯 (Benjamin J. Evans) , (英) 詹姆斯·高夫 (James Gough) , (英) 克里斯·纽兰 (Chris Newland) 著 ; 曾波译. -- 北京 : 人民邮电出版社, 2021.1
（图灵程序设计丛书）
ISBN 978-7-115-55233-4

Ⅰ. ①J… Ⅱ. ①本… ②詹… ③克… ④曾… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2020)第220333号

内 容 提 要

本书从实验科学的角度探讨了 Java 性能优化的方方面面，重点阐述了最新的实用 JVM 性能调优策略、工具和技巧。通过本书，我们不仅可以了解 Java 原理和技术如何充分利用现代硬件和操作系统、衡量 Java 性能的陷阱以及微基准测试的弊端有哪些，还能深入研究可能使团队烦恼的几种性能测试和常见反模式、JVM 垃圾收集、JIT 编译和 Java 语言性能技术等。本书为读者打开了深入理解 Java 性能的大门，并以路标助力其优化之旅。

本书适合 Java 开发人员阅读。

◆ 著　[英] 本杰明·J. 埃文斯　詹姆斯·高夫　克里斯·纽兰
译　曾　波
责任编辑　张海艳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：21.25
字数：502千字　2021年 1 月第 1 版
印数：1－3 000册　2021年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2020-4446 号

定价：109.00元
读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

谨以此书献给我的妻子 Anna，她不仅为本书绘制了精美的插图，还编辑了部分内容，更为重要的是，她经常是倾听我想法的第一人。

——本杰明 · J. 埃文斯

谨以此书献给我亲爱的家人 Megan、Emily 和 Anna。没有他们的帮助和支持，我不可能写出本书。感谢我的父母 Paul 和 Heather，是他们鼓励我学习并给予不断的支持。

还要感谢本杰明 · J. 埃文斯对我的指导和友谊，很高兴能与他再次合作。

——詹姆斯 · 高夫

谨以此书献给我的妻子 Reena，感谢她对我工作的支持和鼓励；也献给我的儿子 Joshua 和 Hugo，愿他们带着好奇心成长。

——克里斯 · 纽兰

目录

中文版推荐序一

说起来，我和本书的作译者倒是都有些缘分。

我和本书的第一作者本杰明 · J. 埃文斯曾有过一面之缘。那是在 2014 年，我第一次参与 QCon 全球软件开发大会（上海站）的策划。当时我想邀请一位在 Java 领域比较有影响力的技术专家来做主题演讲，于是辗转联系到了埃文斯先生。他非常友善，第一时间表示愿意来分享。他的演讲"JIT 编译和 Hotspot JVM 内部探秘"给参会者留下了深刻印象。此后，我在 InfoQ 上也翻译过他的多篇技术文章。

本书（英文版）出版之后，我第一时间就关注了。听说我的好朋友曾波要担纲翻译，我也非常期待。在他翻译过程中，我有幸提前看到了译稿，也跟他做了很多沟通。整体读下来，能感受到他的用心。因为我也翻译过几本书，所以颇能体会译者的辛苦。从一条条更新记录到与编辑反复沟通，都能体现本书译者的认真和负责。

我感觉本书的最大价值在于作者们所要传达的理念。在性能调优方面，没有任何灵丹妙药。任何性能决策，都要通过合理的性能测试来检验。本书不止一次提到，读者不能盲目使用书中结论，如果要在自己的项目中应用，必须亲自通过实验来检测。

本书有些章节经常让人会心一笑，比如在第 4 章讲到性能反模式的地方，有一条叫"按照坊间传说调优"。大家可能有过这种经验，时不时就会在朋友圈看到有人转发"优化 Java 的 xxx 条规则"，诸如"尽量重用对象""尽量用 final"之类，倒是简单好记，但细究起来，很多是十多年前的"野路子"，没有明确的适用范围，仿佛放之四海而皆准，可叹这种坊间传说还颇有市场。希望本书能够荡涤这种风气。

本书提到的很多性能工具也是读者应该重点关注的。像 JITWatch 以及各种剖析工具，本书分门别类给出了介绍和建议，对于读者深入理解 JVM 和定位性能问题的根源都极具价值。建议读者认真研究，也将它们放入自己的工具箱中，使其成为性能调优的利器。

纸上得来终觉浅，绝知此事要躬行。期待本书能给大家带来真正的价值。

——臧秀涛，InfoQ 技术大会主编

中文版推荐序二

为了维护方便，我的团队刚刚用 Java 重写了唯品会一个顶级性能的 C++ 应用，因为是扛流量的关键节点，原版单机 9 万请求 / 秒已经是颇为瞩目的数字，结果翻写的版本竟然反超了 30% 有余，达到了 13 万请求 / 秒，一举扭转了往常 C++ 程序员对 Java 程序员的惯性鄙视链。

正在兴头上，收到了曾波翻译的《Java 性能优化实践：JVM 调优策略、工具与技巧》译稿。说起同类题材的书，上一本我觉得讲得面面俱到的是臧秀涛翻译的《Java 性能权威指南》，但其成于 2016 年，内容难免有些过时。本书则是一本内容更全面、更现代，讲解更深入的接力者。

如果光看书名，特别是英文版书名，大家可能会觉得这是一本 recipes、cookbooks、tips、tricks 以及各种零零碎碎解决特定问题的知识点大杂烩，但翻开目录，可以看出 3 位合著者很用心地给大家展示了一个关于 Java 性能优化的更完整的知识框架。

借用书中的一句话："要成为一名赛车手，你不必成为工程师，但是一定要有机械共鸣。""机械共鸣"这一说法来自伟大的赛车手 Jackie Stewart，他曾 3 次获得世界汽车联合会一级方程式锦标赛冠军。他相信最佳车手对机械如何工作有足够的理解，所以能与赛车协调一致。你不必熟读 *The Java Language Specification*，不必成为 R 大（R 大在我们这群人中就是神），但对于 JVM，必须知道它如何编译、如何运行、如何垃圾收集。

本书除了对原理进行描述外，还对工具做了介绍。从我见过最详尽的 JMH 介绍到 jitwatch，甚至 jHiccup 都在本书中出现了，而在看到作者们列出 JProfiler 和 Yourkit 后，将它们大手一挥归到传统 Profiler 中去，然后 Async Profiler 的登场，让我放下了心。

祝每位 Java 程序员读完本书后，都一边享受 Java 成熟的类库，一边感受 JIT 不输于 C++/Rust 们 AOT 预编译的硬气，一边赞美最新 JDK 的垃圾收集工具 ZGC，一边等待 JVM 协程项目 Loom 的落地，事有不谐时一起拿出火焰图工具 Async Profiler 和线上诊断工具 Arthas 找瓶颈，最后都写出自己性能灿烂的代码。

——江南白衣（肖桦），唯品会架构专家

序

你如何定义性能?

当被问及应用程序的性能时，大部分开发人员会假定他们需要测量某些速度值，比如每秒交易数，或者处理了多少吉字节（GB）数据……，要在尽可能短的时间里完成大量工作。如果你是应用程序架构师，那你可能会测量更广泛的指标。与按直线逻辑执行的程序相比，你或许更关注资源利用率。你可能更重视服务间连接的性能，而不是服务本身的性能。如果你要为公司做出业务决策，应用程序的性能很多时候不是用时间而是用美元来计算的。你可能会与开发人员和架构师争论资源分配，权衡 DevOps 的成本和完成公司工作所需要的时间。

不管你认同哪种角色，所有这些指标都很重要。

我从 1996 年开始开发 Java 应用程序。那时我刚辞掉第一份工作——在明尼苏达大学商学院编写 AppleScript CGI，转而为 Web 开发团队维护服务端 Perl 应用程序。当时 Java 还是非常新的东西，当年年初才发布第一个稳定版本 1.0.2。我接手的任务是开发点有用的东西。

在那个时候，要提升 Java 应用程序的性能，最好的方法是用其他语言重写。那时 Java 还没有即时编译器（JIT compiler），也没有并行和并发垃圾收集器，离 Java 技术统治服务端还早得很。但是我们中的很多人想使用 Java，因此我们尝试了各种技巧来使代码良好地运行。我们编写巨大的方法，以避免方法分派（dispatch）成本。因为垃圾收集很慢，而且有破坏性，所以我们使用了对象池并复用对象。我们还使用了大量的全局状态和静态方法。我们编写的是非常糟糕的 Java 代码，但就当时而言是有效的。

1999 年，情况开始改变。

任何对速度有要求的地方，我们都要与 Java 缠斗不休，如此多年之后，JIT 技术向我们走来了。有了内联方法的编译器，与将巨型组件分解成多个小块相比，方法调用的数量已经不那么重要。我们愉快地拥抱面向对象的设计，将方法拆分成小段，围绕各种事物封装接口。因为我们正在编写良好的 Java 代码，而且 JIT 编译器喜欢这种代码，所以我们对 Java 的每个版本都在提升运行速度感到惊奇。Java 赶超了服务器上的其他技术，使得我们可以用更丰富的抽象构建越来越大、越来越复杂的应用程序。

与此同时，垃圾收集器也迅速改进。现在再使用池化技术往往不如再分配来得划算。很多垃圾收集器提供了多线程操作，我们开始看到低延迟、近乎不影响应用程序的并发垃圾收集器。标准实践朝着开发人员可以放心大胆地创建和丢弃对象的方向发展，并承诺足够聪明的垃圾收集器会处理好这一切。至少就当时而言这是有效的。

技术的问题是，它总是在革自己的命。随着 JIT 和垃圾收集技术的改进，优化应用程序性能的路径也越来越难以捉摸。即使 JVM 可以优化我们的代码，并且让对象几乎没什么成本，但应用程序和用户的需求也在持续增长。

有些时候，甚至是大部分时候，“好的”编码模式盛行：小方法会恰当内联，接口和类型检查成本变低，JIT 编译器生成的原生代码紧凑又高效。但是其他时候，考虑到编译器和 CPU 的限制，我们需要手动调整代码，改变抽象和架构。有些时候，对象几乎是没什么成本的，都不用考虑我们会消耗内存带宽和垃圾收集周期。其他时候，我们要处理 TB 甚至更大规模的数据集，这时候即使是最好的垃圾收集器和内存子系统，也要承受很大压力。

而现在，性能问题的答案是了解你的工具。通常这意味着你不但要了解 Java 语言是如何工作的，还要知道 JVM 类库、内存、编译器、垃圾收集器和应用程序运行所在的硬件是如何交互的。在我从事 JRuby 项目的工作中，我学到一个有关 JVM 的不变的真理：所有的性能问题都没有单一的解决方案，而是有很多解决方案。技巧就是找到那些方案，并把最能满足要求的拼凑起来。现在你有了一个应对性能之战的秘密武器，就是你要阅读的这本书。

朋友，翻过这一页，去发现可以使用的工具和技术吧，那是宝贵的财富。学习如何平衡应用程序的设计和可用的资源，如何监控和调优 JVM，如何利用比老旧的类库和模式更高效的最新 Java 技术，如何让 Java 运行如飞！

对 Java 开发人员而言，这是一个激动人心的时刻，从来没有这么多机会在 Java 平台上构建高效、响应式的应用程序。让我们开始吧。

——Charlie Nutter，红帽中间件首席软件工程师

前言

排版约定

本书使用如下排版约定。

黑体字

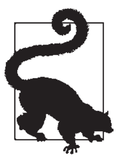
　　表示新术语和重点强调的内容。

等宽字体（`constant width`）

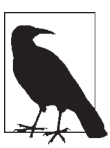
　　用于程序清单，以及在段落内表示程序元素，比如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

带尖括号的等宽字体（`<constant width>`）

　　表示应该使用用户提供的值或根据上下文确定的值来替换的文本。



此图标代表提示或建议。



此图标代表一般性说明。



此图标代表警告或提醒。

使用代码示例

书中示例的源代码请到图灵社区本书主页 http://ituring.cn/book/2085“随书下载”处下载。

本书是帮助你完成工作的。一般来说，如果代码示例是本书提供的，你可以在自己的程序或文档中使用。除非大量复制本书代码，否则你不需要联系我们获取授权。例如，使用本书中的几段代码编写一个程序不需要获取授权。销售或者再发布 O’Reilly 图书中代码的光盘，则需要获取授权。引用本书以及示例代码来回答问题不需要获取授权。将本书中的大量示例代码整合到你的产品文档中，则需要获取授权。

在使用代码时，我们希望能标明出处，但并不强求。出处一般包括书名、作者、出版商和 ISBN。例如，“*Optimizing Java* by Benjamin J. Evans, James Gough, and Chris Newland (O’Reilly). Copyright 2018 Benjamin J. Evans, James Gough, and Chris Newland, 978-1-492-02579-5”。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

O’Reilly在线学习平台（O’Reilly Online Learning）

O'REILLY®

近 40 年来，O’Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O’Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O’Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 http://oreilly.com。

联系我们

与本书有关的评论和问题，请发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

本书有一个页面，可以在那儿找到本书的勘误[1]、示例代码和其他信息：https://www.oreilly.com/library/view/optimizing-java/9781492039259/。

与本书有关的评论或技术问题，请发送电子邮件至 bookquestions@oreilly.com。

要了解更多我们出版的图书、课程、会议和新闻，请访问 http://www.oreilly.com。

注 1：读者也可以到图灵社区本书页面查看和提交勘误，网址是：ituring.cn/book/2085。——编者注

我们在 Facebook 的地址：http://facebook.com/oreilly。

在 Twitter 上的动态：http://twitter.com/oreillymedia。

在 YouTube 上关注我们：http://www.youtube.com/oreillymedia。

致谢

在此要对很多人提供的宝贵帮助表示感谢。

感谢 Charlie Nutter 为本书作序。

感谢以下专家在技术方面的支持，他们提供了很多在任何地方都找不到的信息和知识：

- Christine Flood
- Chris Seaton
- Kirk Pepperdine
- Simon Ritter
- Monica Beckwith
- David Jones
- Richard Warburton
- Stephen Connolly
- Jaroslav Tulach

感谢如下几位提供的细节和说明，以及对我们的鼓励和建议：

- George Ball
- Steve Poole
- Richard Pollock
- Andrew Binstock

感谢技术评审：

- Michael Hsu
- Dmitry Vyazelenko
- Julian Templeman
- Alex Blewitt

感谢 O’Reilly 团队的以下成员：

- Virginia Wilson
- Susan Conant
- Colleen Cole
- Rachel Monaghan
- Nan Barber
- Brian Foster
- Lindsay Ventimiglia

- Maureen Spencer
- Heather Scherer

电子书

扫描如下二维码，即可购买本书中文版电子版。

第 1 章 明确优化与性能

优化 Java 或其他语言代码的性能经常被视作一种暗黑艺术。性能分析有种神秘感，人们常常将其看作孤独的黑客在绞尽脑汁、深思熟虑之后练就的手艺。（孤独的黑客也是好莱坞最喜欢的关于计算机和操作人员的电影桥段之一。）画面是这样的：一个人能够深入了解某个系统，提出神奇的解决方案，使计算机运行得更快。

影像中经常夹杂这种不幸但常见的情况：软件团队没那么重视性能。进而出现的场景是，只有当系统已经陷入麻烦时，团队才会加以分析。所以也就需要性能“英雄”来救场了。不过现实情况有点不同。

事实是，性能分析是坚实的经验主义和软性的人类心理学的奇异组合。重点在于，一方面是可观测指标的绝对数字，另一方面是最终用户和干系人如何**看待**这些数字。本书其余部分的主题就是如何解决这一明显的悖论。

1.1 关于Java性能的误解

多年来，用 Google 搜索“Java performance tuning”，出现的三篇最热门文章之一是于 1997 年到 1998 年左右发表的文章，这篇文章在 Google 搜索引擎出现的早期就被纳入其索引中。这个页面之所以一直在搜索结果中排在前列，是因为其初始排名比较高，带来了很多访问，而这些访问又反过来提升了其排名。

该页面上提供的建议已经完全过时，不再成立，很多情况下甚至对应用程序有害。然而，因为该页面在搜索结果中的有利位置，很多的开发人员能看到它，所以他们很有可能受到这些糟糕建议的影响。

例如，在很早的 Java 版本中，方法分派（method dispatch）性能很差。作为权宜之计，有些 Java 开发人员提倡避免编写小方法，而建议编写大方法。当然，随着时间的推移，虚方

法分派的性能已经得到极大提升。不仅如此，借助现代 Java 虚拟机（Java virtual machine，JVM），特别是自动内联机制，目前大部分调用点（call site）已经消除了虚方法分派。按照“把所有东西集中到一个方法中”这个建议编写的代码处于很大的劣势，因为它对现代即时（just-in-time，JIT）编译器很不友好。

我们还不知道这个糟糕的建议会给应用程序的性能带来多少不利影响，但这个例子清楚地说明了不使用量化和可验证的方式来优化性能是有害的。这也说明对于互联网上的信息，我们不应尽信。

> Java 代码的执行速度是高度动态的，并且从根本上取决于底层的 Java 虚拟机。老的 Java 代码，即使不重新编译，在比较新的 JVM 上可能也可以更快地执行。

正如你所想象的，出于这个原因（其他原因后面再讨论），本书不是一本代码性能技巧手册。相反，我们会聚焦于好的性能工程所涉及的一系列方面：

- 整个软件生命周期内的性能方法论；
- 适用于性能的测试理论；
- 度量、统计和工具；
- 分析技能（包括系统和数据）；
- 底层的技术与机制。

本书后面会引入一些启发式的代码级优化技术，不过这些技术都附带了开发人员在使用之前应该注意的警告和权衡。

> 在正确理解所给建议的上下文之前，不要直接跳到那些章节就开始具体应用这些技术。如果你缺乏正确的理解，那么盲目应用这些技术可能弊大于利。

总之，就是：

- JVM 没有神奇的开关，让你的代码变得“更快”；
- 没有让 Java 运行更快的“秘诀与技巧”；
- 没有对你隐藏的秘密算法。

随着主题的展开，我们会更详细地讨论这些误解，同时介绍开发人员在做 Java 性能分析时常犯的一些错误和相关问题。还在看吧？很好。我们来谈谈性能。

1.2 Java性能概览

要理解 Java 性能为什么是这样的，我们先来看看 Java 之父 James Gosling 的一句很经典的话：

> Java 是一门蓝领语言。它不是博士论文材料，而是用于工作的语言。

也就是说，Java 一直是一种极其实用的语言。它一开始对性能的态度是，只要环境**足够快**并且能提升开发效率，就可以牺牲原始性能。所以直到近些年，随着诸如 HotSpot 之类的

JVM 日趋成熟和进步，Java 环境才开始适合于高性能计算应用程序。

这种实用性在 Java 平台中以多种方式体现出来，但最明显的一点就是使用了**托管子系统**。其理念是，开发人员无须担心托管环境下的某些功能细节，而代价是放弃对底层的一些控制。

最明显的例子当然是内存管理了。JVM 以可插拔垃圾收集子系统的形式提供自动内存管理，所以程序员不必手动跟踪内存。

> 托管子系统在 JVM 中随处可见，它们的存在给 JVM 应用程序的运行时行为带来了额外的复杂性。

正如下一节将讨论的那样，JVM 应用程序复杂的运行时行为要求我们把应用程序看作处于测试下的实验。这会让我们思考观测到的统计数据，也有了一个不幸的发现。

观测到的 JVM 应用程序的性能数据往往不是正态分布的，这意味着基本的统计技术（比如**标准差**和**方差**）不适合处理来自 JVM 应用程序的结果。这是因为很多基本的统计方法隐含着一个假设，即结果是正态分布的。

一种理解方式是，对于 JVM 应用程序，异常值非常重要，比如低延迟交易应用程序。这意味着测量值的采样也存在问题，因为它很容易错过最重要的精确事件。

最后提醒一句：我们很容易被 Java 性能测量值误导。环境的复杂性意味着很难隔离系统的各个方面。

测量也有成本，频繁采样（或者记录每个结果）可能会对所记录的性能数字造成明显的影响。Java 性能数字的特性要求我们使用一些复杂的统计方法，而将基本的技术应用于 Java/JVM 应用程序时，经常会产生不正确的结果。

1.3 作为实验科学的性能

和大多数现代软件系统一样，Java/JVM 软件栈非常复杂。事实上，因为 JVM 具有高度优化和自适应的特性，所以构建于 JVM 之上的生产系统有时会表现出一些非常微妙且复杂的性能行为。这种复杂性和摩尔定律以及该定律所表示的硬件能力的空前增长有关。

> 计算机软件产业最为惊人的成就是其持续不断地抵消了硬件产业所取得的稳定而巨大的成果。
>
> ——Henry Petroski

尽管有些软件系统浪费了硬件产业过去的成果，但是 JVM 所代表的是某种工程上的胜利。从 20 世纪 90 年代末诞生以来，JVM 已经发展为一个高性能的、通用的执行环境，能够很好地利用新的硬件特性。但是，与任何复杂的高性能系统一样，JVM 需要我们有相当程度的技能和经验才能从中获得最大的收益。

> 如果不能清晰地定义测量，那比无效还要糟糕。
>
> ——Eli Goldratt

因此，JVM 性能调优是技术、方法论、可测的量和工具的综合。它的目的是以系统所有者或用户所期望的方式产生可以测量的输出。换句话说，性能是一门实验科学，它通过以下方式实现预期结果：

- 定义期望的结果；
- 测量现有系统；
- 确定要实现需求所需的工作；
- 开始某个改进操作；
- 重新测试；
- 确定目标是否实现。

定义和确定预期性能结果的过程创建了一组量化的目标。确定应该测量什么并记录这些目标非常重要，而它们将成为项目工件和可交付成果的一部分。由此我们可以看到，性能分析是建立在定义和实现非功能性需求的基础之上的。

正如前面提到的，这个过程可不是占卜。相反，我们依赖于统计数据和对结果的适当处理。第 5 章将介绍基本的统计技术，用于精确处理某个 JVM 性能分析项目生成的数据。

对于很多实际的项目而言，无疑需要对数据和统计有更深入的理解。建议你把本书中出现的统计技术当作起点，而不是说有这些就够了。

1.4 性能分类方法

本节将介绍一些基本的性能指标，它们为性能分析提供了一个词汇表，让你可以量化调优项目的目标。这些目标就是定义性能目标的非功能性需求。一些基本的性能指标如下：

- 吞吐量（throughput）
- 延迟（latency）
- 容量（capacity）
- 利用率（utilization）
- 效率（efficiency）
- 可扩展性（scalability）
- 降级（degradation）

下面将依次简要地探讨这些指标。注意，对大部分性能项目而言，并非要同时优化每个指标。在每次性能迭代中只改进几个指标的情况更为常见，而且一次可以调优的指标可能也就这么多。在实际项目中，很可能会出现这样的情况：优化一个指标可能会损害另外一个或几个指标。

1.4.1 吞吐量

吞吐量是一个表示系统或子系统能够执行的工作速率的指标，通常用某个时间段内的工作单元数来表示。比如，我们可能会对系统每秒能执行多少个事务感兴趣。

要使吞吐量数字在实际的性能实践中具有意义，那它应该包括对获得该数据的参考平台的

描述。比如，硬件规格、操作系统和软件栈都与吞吐量有关；被测试的系统是单台服务器还是集群也是如此。此外，不同测试之间的事务数（或工作单元）应该是相同的。基本上，我们应该设法确保进行吞吐量测试的工作负载在多次运行之间保持一致。

1.4.2　延迟

有时可以把性能指标比喻为水管，以此来解释它。如果一根水管每秒能流出 100 升水，那么 1 秒内流出的水量（100 升）就是吞吐量。在这个比喻中，延迟实际上就是水管的长度。也就是说，它是处理一个事务并在水管的另一端看到结果所花费的时间。

它通常被称为端到端的时间。它取决于工作负载，所以一个比较常见的方法是制作一张图，将延迟显示为不断增加的工作负载的函数。我们将在 1.5 节看到这类图。

1.4.3　容量

容量是指系统所拥有的工作并行度，即系统中可以同时进行的工作单元（比如事务）的数量。

容量显然与吞吐量有关，而且我们应该可以预料到，随着系统上并发负载的增加，吞吐量（和延迟）会受到影响。因此，容量通常被看作在给定的延迟或吞吐量值下可用的处理能力。

1.4.4　利用率

最常见的性能分析任务之一就是实现系统资源的高效利用。理想情况下，CPU 应该用于处理各个工作单元，而不是闲置（或者花时间处理操作系统或其他管理任务）。

根据工作负载的不同，不同资源的利用水平可能差异巨大。比如，在计算密集型的工作负载（如图形处理或加密）运行的时候，CPU 利用率可能接近 100%，但只使用了一小部分可用内存。

1.4.5　效率

将系统的吞吐量除以所使用的资源可以衡量系统的整体效率。直观上看，这是有道理的，因为要获得同样的吞吐量，需要更多资源的那个系统可以被定义为效率低下。

在处理较大的系统时，也可以使用成本核算的形式来衡量效率。如果用美元计算，解决方案 A 的总体拥有成本（TCO）是解决方案 B 的两倍，那么在相同的吞吐量下，A 的效率显然是 B 的一半。

1.4.6　可扩展性

系统的吞吐量或容量取决于可用于处理的资源。添加资源时，吞吐量的变化是衡量系统或应用程序可扩展性的一个标准。系统可扩展性的最理想情况是，吞吐量的变化和资源的变化步调一致。

考虑一个基于服务器集群的系统。如果集群被扩大，比如说规模扩大了一倍，那么可以实

现什么样的吞吐量呢？如果新的集群可以处理两倍的事务量，那么该系统将呈现“完美的线性扩展”。这在实践中很难实现，特别是在面对各种可能的负载时。

系统的可扩展性取决于很多因素，而且通常不是一个简单的恒定因素。常见的情况是，对于某个范围内的资源，系统的可扩展性接近于线性，但在更高的负载下，系统会遇到一些阻碍完美扩展的限制。

1.4.7 降级

如果我们通过增加请求（或客户端）的数量或加快请求到达的速度来增加系统的负载，可能会看到观察到的延迟或吞吐量的变化。

请注意，这种变化取决于系统的利用率。如果系统利用率不足，那么在出现可观测到的变化之前，系统应该比较平缓；但如果资源已经被充分利用，那么我们会看到吞吐量停止增加或延迟增加。这些变化通常被称为系统在额外负载下的降级。

1.4.8 各种性能观测之间的关联

各种性能观测行为通常会以某种方式关联在一起。这种关联的细节取决于系统是否在峰值利用率下运行。比如，一般情况下，利用率会随着系统负载的增加而改变。然而，如果系统的利用率不足，那么增加负载未必会明显提高利用率。反之，如果系统已经处于压力之下，那么增加负载的效果可能就会在另一个观测中感受到。

另一个例子，可扩展性和降级都代表着系统行为在负载增加时的变化。对于可扩展性来说，随着负载的增加，可用资源也会增加，但核心问题是系统能否利用这些资源。然而，如果负载增加了，却没有提供额外的资源，那么有些可观测到的性能降级（比如延迟）就是可以预料到的结果了。

> 在极少数情况下，额外的负载会引发违反直觉的结果。比如，如果负载的变化导致系统的某些部分切换到资源更密集但性能更高的模式，那么即使接收到更多的请求，总体效果反而是延迟降低。

例如，第 9 章将详细讨论 HotSpot 的 JIT 编译器。要想进行 JIT 编译，方法必须在解释模式下“足够频繁地”执行。因此，在低负载时，有些关键方法可能会卡在解释模式下，但在较高负载时，由于对方法的调用频率增加，这些方法就有资格被编译。这就导致以后对同一方法的调用比之前执行的速度快得多。

不同的工作负载会有截然不同的特性。比如金融市场上的一笔交易，从头到尾看，其执行时间（即延迟）可能是几个小时甚至几天。然而，在任何特定的时间下，一家大银行可能有数百万笔此类交易正在进行。因此，系统的容量很大，但延迟也很大。

然而，我们现在只考虑银行内部的一个子系统。买方和卖方的匹配（本质上是双方商定价格）被称为**对盘**（order matching）。这个单独的子系统在任何特定的时间可能只有数百个挂单（pending order），但从接受订单到完成匹配的延迟可能只有 1 毫秒（在“低延迟”交

易的情况下甚至更短）。

本节介绍了最常见的性能观测指标。尽管用到的定义偶尔可能会稍有不同，甚至指标也有所差别，但在大部分情况下，它们通常是指导我们进行系统调优的基本的系统数字，也是探讨我们所感兴趣的系统性能时的分类依据。

1.5　阅读性能图

最后来看看性能测试中出现的一些常见行为模式。我们将查看一些来自真实观测结果的图，在进行过程中还会遇到很多其他数据图表的例子。

图 1-1 显示了随着负载的增加，性能（本例中是延迟）出现了突然的、预料之外的降级，这通常叫作**性能拐点**（performance elbow）。

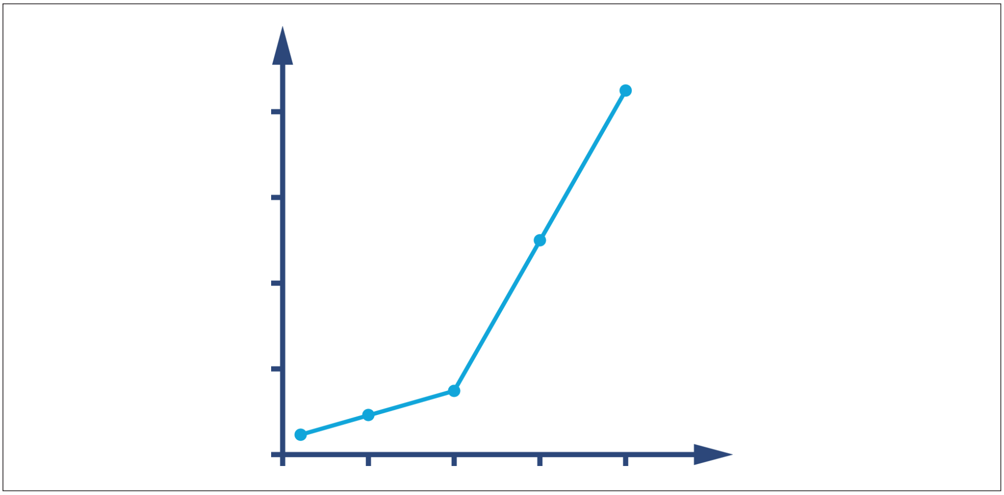


图 1-1：性能拐点

相比之下，图 1-2 显示的情况更让人满意，随着集群中不断添加机器，吞吐量几乎呈线性扩展。这很接近于理想的行为，而且只会在极佳的情况下才有可能实现，比如，扩展的是与某台单一服务器没有会话亲和性的无状态协议。

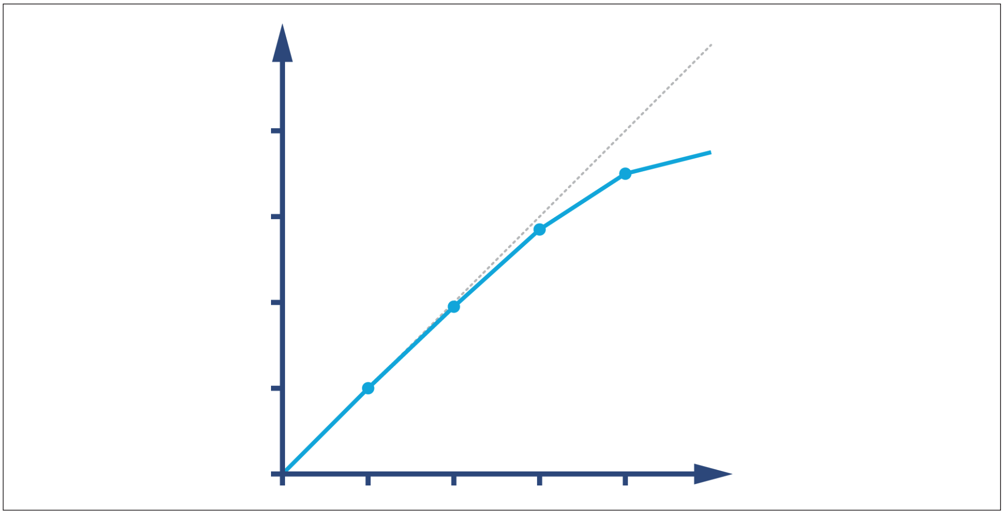


图 1-2：近线性扩展

第 12 章将介绍 Amdahl 定律，该定律以 IBM 公司的著名计算机科学家（也是“大型机之父”）Gene Amdahl 的名字命名。图 1-3 就是 Amdahl 对可扩展性的基本约束的示意图，它显示了最大可能的速度提升是专用于该任务的处理器数量的函数。

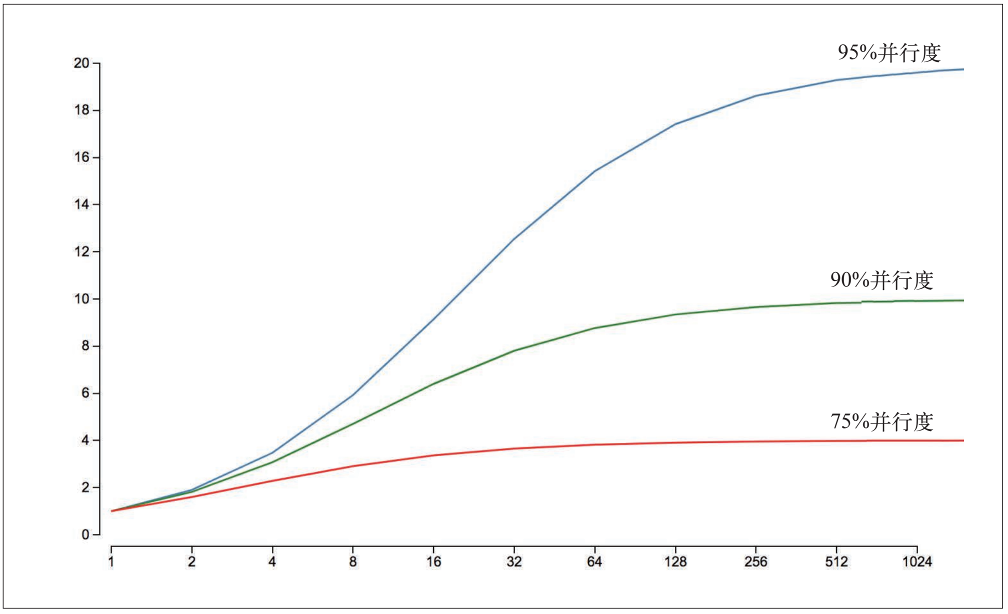


图 1-3：Amdahl 定律

我们展示了 3 种情况：底层任务的并行度分别为 75%、90% 和 95%。这清楚地表明，只要工作负载有任何部分必须串行执行，线性可扩展性就不可能实现，而且对能够实现多大程

度的可扩展性有严格的限制。这就证明了围绕图 1-2 的说明是正确的——即使情况良好，线性可扩展性也不可能实现。

Amdahl 定律所施加的限制是非常严格的。特别要注意的是，图中的横轴是对数，因此即使是（只有）5% 串行的算法，也需要 32 个处理器来实现 12 倍的加速。更糟糕的是，无论使用多少个核心，对于该算法来说，最大的提速也只能达到 20 倍。在实际应用中，很多算法的串行度远远超过 5%，所以最大可能的提速倍数就更受约束了。

正如将在第 6 章中看到的，对于没有承受压力的健康应用程序，JVM 垃圾收集子系统中的底层技术会自然地产生一种“锯齿形”的内存模式。示例如图 1-4 所示。

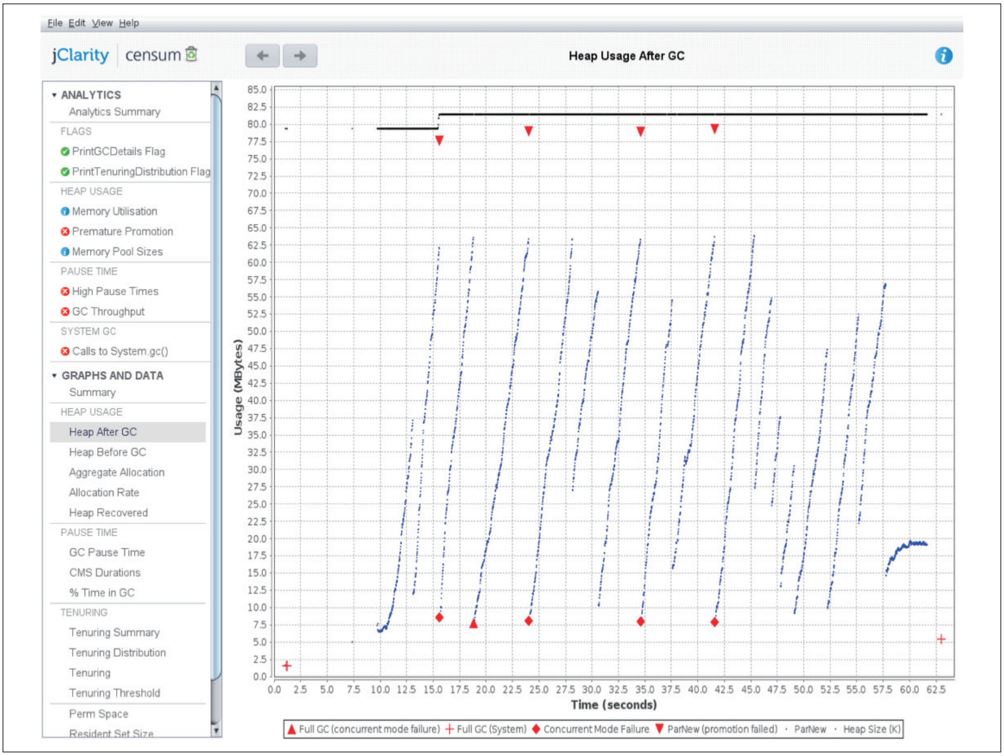


图 1-4：健康的内存使用情况

图 1-5 显示了另一个内存图，在对应用程序的内存分配率进行性能调优时，这个图非常重要。这个简短的例子显示的是一个示例应用程序（计算 Fibonnaci 数），它清楚地显示出分配率在 900 秒左右有个暴跌。

来自同一工具（jClarity Censum）的其他图表显示，应用程序此时开始出现重大的垃圾收集问题，进而，由于垃圾收集线程争用 CPU 资源，应用程序无法分配足够的内存。

我们发现分配子系统的运行也是一个热点，每秒分配的内存超过 4 GB，远高于大多数现代系统（包括服务器级硬件）所推荐的最大容量。关于内存分配这个主题，在第 6 章介绍垃圾收集时会进行更多的讨论。

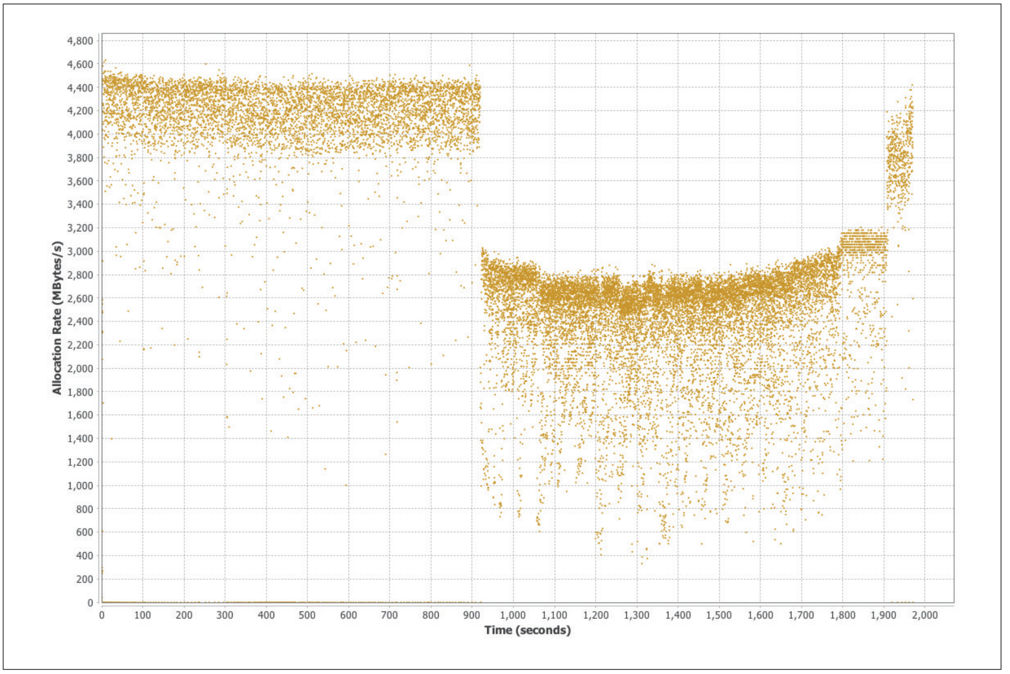


图 1-5：有问题的内存分配率示例

系统出现资源泄漏时，通常以图 1-6 所示的方式表现出来。在这种情况下，观测量（在这个例子中是延迟）会随着负载的增加而缓慢下降，在达到一个拐点后，系统会迅速降级。

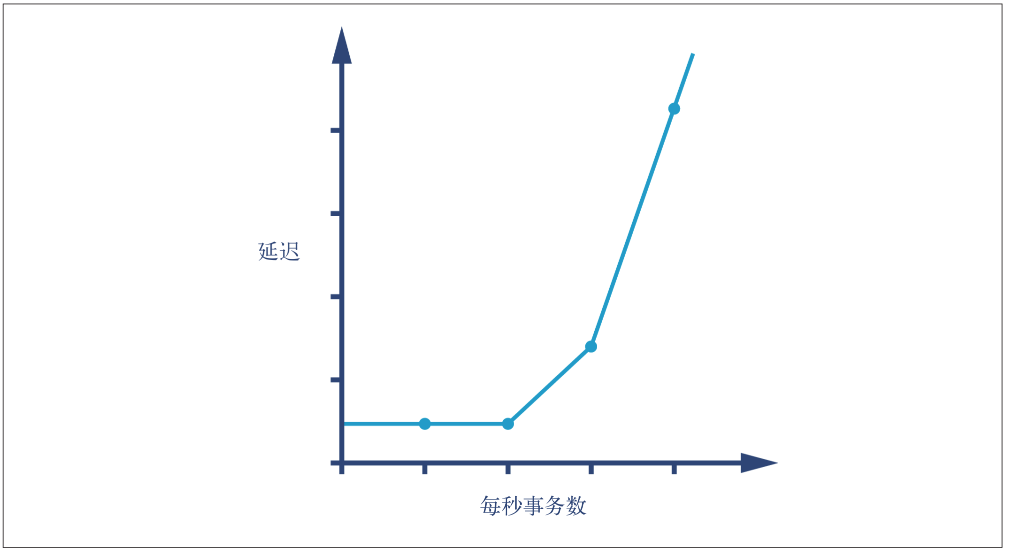


图 1-6：高负载下的延迟

1.6 小结

本章首先讨论了 Java 的性能是什么，不是什么；然后介绍了经验科学和测量的基本主题，以及一个好的性能实践将用到的基本词汇和观测量；最后介绍了性能测试结果中一些常见的案例。接下来我们将开始讨论 JVM 的一些主要内容，并为理解到底是什么导致基于 JVM 的性能优化如此复杂做好准备。

第 2 章

JVM概览

Java 无疑是地球上最大的技术平台之一，根据 Oracle 的数据，该平台拥有大约 900 万到 1000 万的开发人员。按照设计，很多开发人员不需要了解平台底层的复杂机制。这就导致一种情况，只有当客户抱怨性能时，他们才会遇到这些细节问题。

不过对于关心性能的开发人员而言，理解 JVM 技术栈的基础非常重要。了解 JVM 技术可使开发人员编写出更好的软件，并为调查与性能相关的问题提供必要的理论背景。

本章将介绍 JVM 如何执行 Java，为后面章节更深入地探索这些主题打下基础。特别是第 9 章会深入介绍字节码。读者可以选择现在阅读本章，也可以在理解了其他主题之后结合第 9 章一起重读。

2.1 解释和类加载

根据 Java 虚拟机规范（VM Spec），JVM 是基于栈的解释型机器。这意味着 JVM 和物理硬件 CPU 不同，它没有寄存器，而是使用一个包含部分结果的执行栈，并通过操作该栈顶的一个或多个值来执行计算。

可以把 JVM 解释器的基本行为理解为一个“包含在 `while` 循环中的 `switch` 语句”（switch-inside-while），按顺序单独处理程序的每一个操作码，使用求值栈保存中间值。

> 当深入研究 Oracle/OpenJDK VM（HotSpot）内部时会看到，真正的产品级 Java 解释器情况可能更加复杂，不过目前 switch-inside-while 这种模式更容易接受。

当使用 `java HelloWorld` 命令启动应用程序时，操作系统会启动 Java 虚拟机进程（`java` 二进制程序）。它会设置 Java 虚拟环境，并初始化这一栈式机器，而这一机器将真正执行

HelloWorld 类文件中的用户代码。

应用程序的入口点是 `HelloWorld.class` 的 `main()` 方法。虚拟机执行之前必须先加载这个类，然后才能将控制权交给它。

要实现这一点，需要使用 Java 的类加载机制。当启动一个新的 Java 进程时，会用到一个加载器链。初始的加载器就是所谓的启动类加载器（bootstrap classloader），它包含核心 Java 运行时中的类。在 Java 8 及之前的版本中，它们都是从 `rt.jar` 中加载的。而在 Java 9 及以后的版本中，运行时已经被模块化，类加载的概念有些差别。

启动类加载器的要点是获得类的一个最小集合（包括诸如 `java.lang.Object`、`Class` 和 `Classloader` 等核心类），以允许其他类加载器启动系统的其他部分。

> Java 将类加载器建模为其运行时和类型系统内的对象，所以需要以某种方式将一组初始类加载进来。否则，在定义类加载器时就会出现循环问题。

下一步创建的是扩展类加载器（extension classloader）。它将启动类加载器定义为其父加载器，并在需要时将加载工作委托给启动类加载器。扩展类加载器的应用并不广泛，但是可以为具体的操作系统和平台提供重写代码和原生代码。值得注意的是，Java 8 中引入的 Nashorn JavaScript 运行时就是由扩展类加载器来加载的。

最后创建的是应用类加载器（application classloader）。它负责从指定的类路径（classpath）中加载用户类。很遗憾，有些文章称其为“系统”类加载器（system classloader），我们应该避免这种叫法，原因很简单，它并没有加载系统类（这是启动类加载器的工作）。应用类加载器最为常见，它的父加载器是扩展类加载器。

当在程序执行过程中遇到对新类的依赖时，Java 会加载它们。如果一个类加载器没有找到某个类，则加载行为将被委托给其父加载器。如果链式查找已经到了启动类加载器这里，但是还没找到它，则抛出 `ClassNotFoundException` 异常。要缓解这一问题，非常重要的一点是对于与生产中使用的完全相同的类路径，开发人员应该使用一个能高效编译的构建过程。

正常情况下，Java 只会对一个类加载一次，并在运行时环境中创建一个 `Class` 对象来表示它。不过需要注意的是，同一个类有可能被不同的类加载器加载两次。因此，类在系统中是通过加载它的类加载器以及其全限定类名（包括程序包名）来识别的。

2.2 执行字节码

重要的是要认识到，Java 源代码在执行之前要经历一系列变换。首先就是使用 Java 编译器 `javac` 进行的编译阶段，`javac` 通常作为更上层构建流程的一部分调用。

`javac` 的工作是将 Java 代码转换为包含字节码的 .class 文件。它对 Java 源代码做了非常直接的变换，如图 2-1 所示。 `javac` 在编译期间很少进行优化，而且以反汇编工具（比如标准的 `javap`）查看其生成的字节码时，可读性仍然非常好，可以像 Java 代码那样识别。

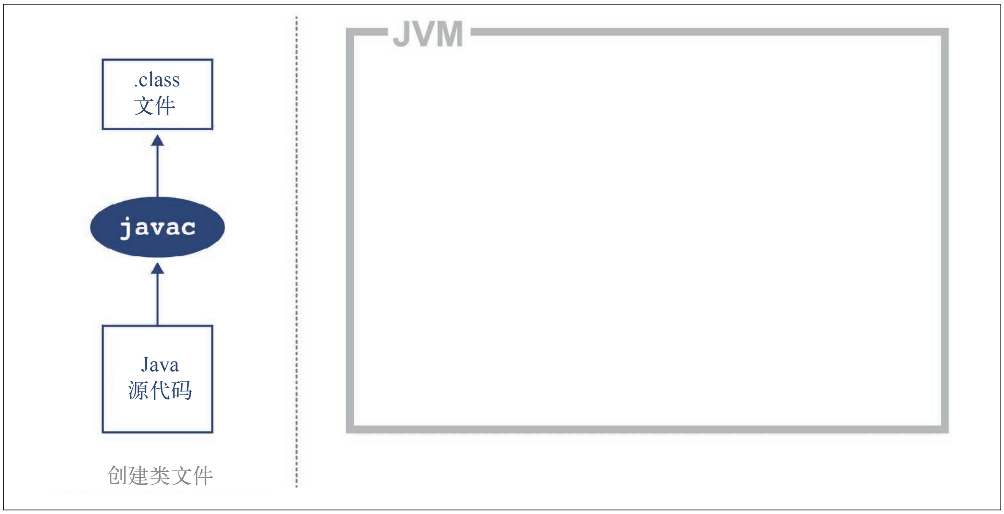


图 2-1：Java 类文件编译

字节码是一种中间表示，没有与特定的机器架构绑定。与机器架构解耦有两大好处：一是提供了可移植性，这意味着已经开发好或编译完的软件可以在 JVM 支持的任何平台上运行；二是提供了对 Java 语言的抽象。这为我们了解 JVM 执行代码的方式提供了第一个重要视角。

> 现在，Java 语言和 Java 虚拟机在一定程度上是独立的，所以 JVM 这个叫法可能有点误导，因为 JVM 可以执行任何能生成合法类文件的 JVM 语言。事实上，图 2-1 也很容易演示 Scala 编译器 `scalac` 生成字节码并在 JVM 上执行的逻辑。

不管使用的是何种源代码编译器，生成的类文件都具备虚拟机规范所指定的明确结构（参见表 2-1）。JVM 对于任何要加载的类，都要先验证它们符合指定格式，然后才允许其执行。

表2-1：类文件剖析

组　件	描　述
魔数	`0xCAFEBABE`
类文件格式版本	该类文件的主版本号和次版本号
常量池	该类的常量池
访问标志	该类是否为抽象类、静态类等
当前类	该类的名字
超类	超类的名字
接口	该类实现的接口
字段	该类的所有字段
方法	该类的所有方法
属性	该类的所有属性（比如，源文件的名字，等等）

每个类文件都以魔数（magic number）`0xCAFEBABE` 开始，前面的 4 个以十六进制表示的字节表示当前文件符合类文件格式。随后的 4 个字节表示用于编译该类文件的主版本号和次版本号，检查这些字节可以确保目标 JVM 的版本不低于编译该类文件的版本。类加载器会检查主版本号和次版本号，以确保兼容性；如果不兼容，则在运行时抛出 `UnsupportedClassVersionError`，说明运行时的版本低于所编译类文件的版本。

> 魔数为 Unix 环境提供了一种识别文件类型的方式（而 Windows 通常使用文件扩展名）。为此，这个值一旦决定就很难修改了。遗憾的是，尽管 Java 9 为模块文件引入了魔数 `0xCAFEDADA`[1]，但在可预见的未来，Java 仍然不得不继续使用令人相当尴尬和存在性别歧视的 `0xCAFEBABE`。

常量池（constant pool）保存代码中的常量，比如类、接口和字段的名字。当 JVM 执行代码时，会使用常量池表来查找值，而不用依赖运行时的内存布局。

访问标志（access flag）用于确定应用于该类的修饰符。标志的第一部分识别一般属性，比如一个类是否为 `public`，随后的部分判断该类是否为 `final`（如果是，则不能子类化）。这个标志还可以确定该类文件表示的是接口还是抽象类。标志的最后一部分指示这个类文件是表示源代码中没有出现的合成类、注解（annotation）类型或枚举（enum）。

当前类（`this` class）、超类（superclass）和接口（interface）这几项都是指向常量池的索引，用于识别该类相关的类型层次。字段（field）和方法（method）定义了一个类似签名的结构，包括应用于字段或方法的修饰符。然后使用一组属性，用于表示更复杂、大小不固定的结构中的结构化条目。比如，方法使用 `Code` 属性来表示与该特定方法关联的字节码。

图 2-2 提供了一个辅助方法，帮你记住类文件的结构。

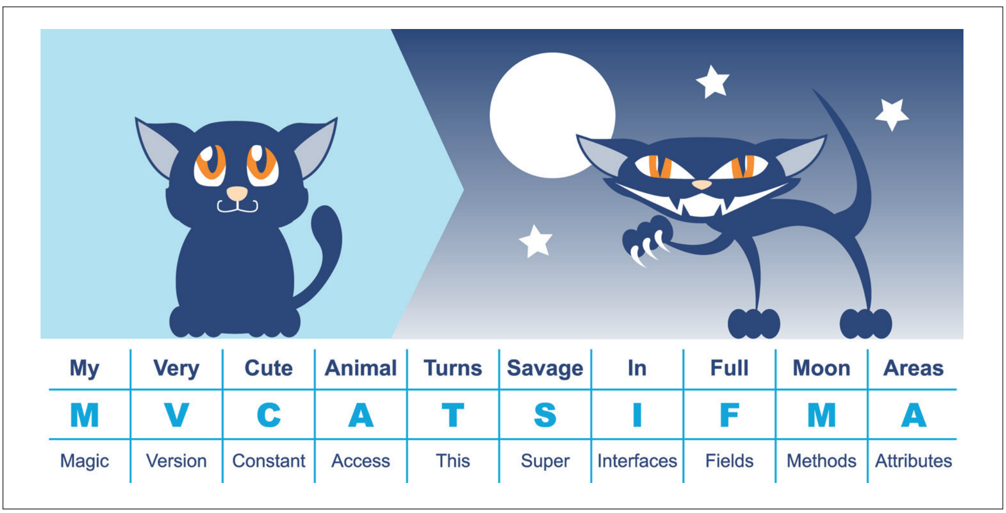


图 2-2：类文件结构助记法

注 1：`0xCAFEDADA` 是 JIMAGE 文件的魔数。——译者注

在这个非常简单的代码示例中，可以观察到运行 javac 的作用：

```
public class HelloWorld {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello World");
        }
    }
}
```

Java 提供了一个名叫 javap 的类文件反汇编器，可用于检视类文件。以 HelloWorld 类文件为例，运行 javap -c HelloWorld，得到如下输出：

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1     // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        10
       5: if_icmpge     22
       8: getstatic     #2     // Field java/lang/System.out ...
      11: ldc           #3     // String Hello World
      13: invokevirtual #4     // Method java/io/PrintStream.println ...
      16: iinc          1, 1
      19: goto          2
      22: return
}
```

上面的输出描述了 HelloWorld.class 文件的字节码。要了解更多细节，javap 还有一个 -v 选项，可以提供完整的类文件头信息和常量池详情。尽管源代码中只有一个 main() 方法，但是这个类文件包含两个方法，这是因为 javac 自动为该类添加了一个默认构造器。

构造器中执行的第一个指令是 aload_0，它将 this 引用放到栈中的第一个位置。然后调用 invokespecial 命令，该命令调用一个实例方法，该方法有特殊处理，会调用超类的构造器并创建对象。在默认构造器中，因为没有覆写，所以调用匹配的是 Object 的默认构造器。

> JVM 中的操作码非常简洁，可以表示类型、操作，以及局部变量、常量池和栈之间的交互。

再来看 main() 方法，iconst_0 将整型常量 0 压到求值栈上。istore_1 将该常量值保存在局部变量表中偏移为 1 的位置（代表循环中的 i）。局部变量偏移从 0 开始，但是对于实例方法，第 0 项总是 this。之后将偏移 1 的变量加载回栈上，并使用 if_icmpge（if integer compare greater or equal）将常量 10 推送到栈中进行比较。只有在当前的整型值不小于 10

时，测试才会成功。

对于前几次迭代，比较测试失败，所以我们继续走到指令 8。这里会解析出 `System.out` 的静态方法，然后从常量池加载“Hello World”字符串。下一个调用，即 `invokevirtual`，会调用该类的一个实例方法。该整型数增加，之后 `goto` 被调用，跳回到指令 2。

这个过程一直持续到 `if_icmpge` 比较最终成功（当循环变量不小于 10 时）；在这次迭代时，控制传递到指令 22，方法返回。

2.3 HotSpot简介

1999 年 4 月，Sun 公司向 Java 引入了性能方面最大的一个变化。HotSpot 虚拟机是 Java 的一个关键特性，使得 Java 的性能可以与 C 和 C++ 比肩（甚至超过），如图 2-3 所示。要解释其原因，我们先来深入了解一下 Java 为支持应用程序开发所做的语言设计。

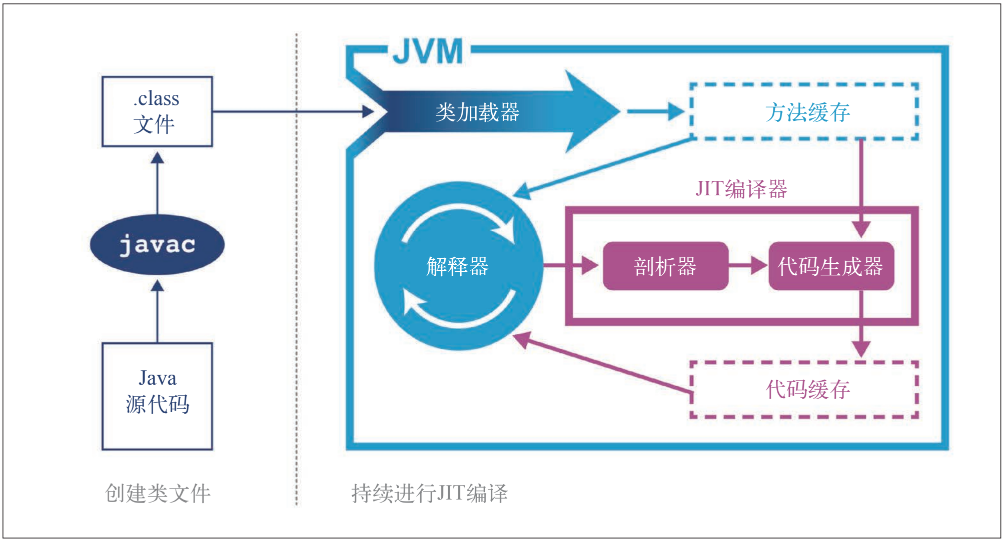


图 2-3：HotSpot JVM

语言和平台设计经常涉及在所需功能之间进行决策和权衡。在设计 Java 时，面对的选择是让它“更接近机器”、依赖诸如“零成本抽象”等原则，还是提高开发人员的效率，首选“把事情做完”而不是严格的底层控制。

> C++ 实现遵循零成本原则：不需要为用不到的功能付出代价。更进一步，你要使用的东西，性能已经非常好，手动优化也不会做得更好了。
>
> ——Bjarne Stroustrup

零成本原则理论上听起来很好，但它要求所有使用这门语言的人都要面对操作系统和计算机如何工作这些底层机制。对于并不将原始性能作为主要目标的开发人员而言，这是一个很大的额外认知负担。

不仅如此，它还要求在构建时将源代码编译为平台特定的机器代码——通常叫作**预先**（ahead-of-time，AOT）**编译**，因为诸如解释器、虚拟机和可移植层之类的执行模式都不是零成本的。

这个原则也隐藏了很多问题，比如“你要使用的东西性能已经非常好，手动优化也不会做得更好了”。这就预设了很多东西，尤其是开发人员能够编写出比自动化的系统更好的代码。这个假设根本不合理。现在几乎没有人再想用汇编语言写代码了，所以使用自动化系统（如编译器）来生成代码显然对大部分程序员有一些好处。

Java 从来就没有认可过零成本抽象哲学。相反，HotSpot 虚拟机所采取的方法是分析应用程序的运行时行为，并智能地在最有可能获得性能提升的地方应用优化。HotSpot 虚拟机的目标是支持你编写符合 Java 的代码，并遵循良好的设计原则，而不是让程序来适合虚拟机。

即时编译简介

Java 程序在字节码解释器中开始执行，而在解释器中，指令是在虚拟化的栈式机器上执行的。这种从真正的 CPU 抽象出来的机制使类文件具有了可移植性，但是为了获得最好的性能，程序还是必须利用 CPU 的原生特性，直接在 CPU 上执行。

HotSpot 通过将程序单元从解释的字节码编译成原生代码来实现这个目标。HotSpot 虚拟机中的编译单元是方法和循环。这就是所谓的**即时**编译。

JIT 编译的工作原理是，当应用程序在解释模式下运行时，监控该应用程序，并观察代码中执行最频繁的部分。在这个分析过程中，系统会捕获程序化的轨迹信息，从而实现更复杂的优化。一旦特定方法的执行超过了一个阈值，剖析器就会对这段特定代码进行编译和优化。

JIT 编译方法有很多优点，一个最主要的优点是它将编译器的优化决策建立在解释阶段收集到的轨迹信息的基础上，从而使 HotSpot 能够做出更明智的优化。

不仅如此，HotSpot 经过了多年的开发，几乎每个版本都增加了新的优化和优势。这意味着在新版本 HotSpot 上运行的任何 Java 应用程序都能够利用虚拟机中提供的新的性能优化，甚至不需要重新编译。



在从 Java 源代码转换成字节码并进行另一个（JIT）编译步骤之后，实际执行的代码与所编写的源代码相比发生了非常大的变化。这是一个很关键的洞察，它将决定我们在处理性能相关研究时采用何种方法。在 JVM 上执行的 JIT 编译代码可能与原始的 Java 源代码看上去完全不一样。

大致的情况就是，像 C++ 这样的语言（以及很有前途的 Rust）性能往往更可预期，但代价是给用户带来了大量底层复杂性。

请注意，“更可预期”不一定意味着“更好”。AOT 编译器生成的代码可能需要在很多不同的处理器上运行，并且可能无法假定某些特定的处理器特性是可用的。

使用剖析制导优化（profile-guided optimization，PGO）的环境，比如 Java，有可能以大部

分 AOT 平台根本无法使用的方式使用运行时信息。这可以改善性能，例如动态内联和优化虚拟调用。HotSpot 甚至可以在虚拟机启动时检测到它所运行的 CPU 的精确类型，而且可以使用这些信息来开启针对特定处理器特性设计的优化（如果有的话）。

> 检测精确处理器能力的技术被称为 **JVM 内部函数**（intrinsic），不要与 `synchronized` 关键字引入的内置锁（intrinsic lock）混淆。

关于 PGO 和 JIT 编译的完整讨论参见第 9 章和第 10 章。

对大多数普通开发人员来说，HotSpot 所采取的复杂方法大有裨益，但是这种权衡（放弃零成本抽象）意味着在高性能 Java 应用程序的具体案例中，开发人员必须非常小心，以避免实际执行 Java 应用程序时陷入“常识性”推理和产生过于简化的思维模式。

> 一般来说，分析一小段 Java 代码的性能（**微基准测试**）实际上比分析整个应用程序的性能要难，而且这是大部分开发人员不应该承担的一项非常专业的工作。第 5 章会探讨这个主题。

HotSpot 的编译子系统是虚拟机提供的两个最重要的子系统之一。另一个是自动内存管理，这是早期 Java 的主要卖点之一。

2.4 JVM内存管理

在诸如 C、C++ 和 Objective-C 这样的语言中，程序员负责管理内存的分配和释放。自己管理对象的内存和生命周期的好处是性能更具确定性，而且我们还能将资源的生命周期与对象的创建和删除绑定在一起。但是这些好处是以巨大的成本为代价的：为保证正确性，开发人员必须能够精确掌控内存。

遗憾的是，几十年的实践经验表明，很多开发人员对内存管理的习惯用法和模式了解甚少。C++ 和 Objective-C 的后续版本使用标准库中的智能指针习惯用法改进了这一点。然而，在创建 Java 时，差劲的内存管理是应用程序错误的一个主要原因。这导致开发人员和管理者更关注花在处理语言特性上的时间，而不是用来交付业务价值的时间。

Java 通过引入一个叫作**垃圾收集**（garbage collection，GC）的进程自动管理堆内存来帮助解决这个问题。简而言之，垃圾收集是一个非确定性的过程，当 JVM 需要分配更多内存时，它会触发以恢复和复用不再需要的内存。

然而，垃圾收集背后的故事并不简单。在 Java 的历史过程中，已经开发并应用了各种垃圾收集算法。垃圾收集付出的代价是：当它运行时，往往会造成**全部停顿**，这意味着当垃圾收集进程运行时应用程序会暂停。通常，这些暂停时间设计得非常短，但是当应用程序承受压力时，暂停时间可能会增加。

垃圾收集是 Java 性能优化的一个主要话题，所以我们会用第 6 章、第 7 章和第 8 章专门讨论 Java 垃圾收集的细节。

2.5 线程和Java内存模型

Java 最大的优势之一就是从第一个版本开始就内置了对多线程的支持。Java 平台允许开发人员创建新的执行线程。比如，Java 8 的语法如下：

```
Thread t = new Thread(() -> {System.out.println("Hello World!");});
t.start();
```

不仅如此，Java 环境和 JVM 一样，本身就是多线程的。这给 Java 程序带来了额外的、无法简化的复杂性，使得性能分析工作变得更加困难。

在大部分主流 JVM 实现中，每个 Java 应用程序线程都精确地对应一个专用的操作系统线程。另一种方式是使用一个共享的线程池来执行所有 Java 应用程序线程，这种方式叫作**绿色线程**。事实证明后面这种方式并没有提供可接受的性能状况，还增加了不必要的复杂性。

> 可以大胆地假设，每个 JVM 应用程序线程背后都有唯一的操作系统线程支持，它会在我们调用 `Thread` 对象的 `start()` 方法时被创建出来。

Java 的多线程实现方式可以追溯到 20 世纪 90 年代末期，它有以下 3 个基本的设计原则。

- Java 进程中的所有线程共享一个单一、公共的垃圾回收堆。
- 某个线程创建的任何对象都可以被任何其他具有该对象引用的线程访问。
- 对象在默认情况下是可变的，也就是说，除非程序员明确使用 `final` 关键字将其标记为不可变的，否则对象字段中保存的值可以被修改。

Java 内存模型（Java memory model，JMM）是内存的一个形式化模型，它解释了不同执行线程如何看到对象中保存的正在修改的值。也就是说，如果线程 A 和线程 B 都有对对象 `obj` 的引用，而线程 A 对其进行了修改，那么在线程 B 中观察到的值会发生什么变化？

这个问题看似简单，但因为操作系统的调度器（将在第 3 章中介绍）可以强行将线程从 CPU 核心中剥离出去，所以事实上要更加复杂。这可能会导致另一个线程开始执行，并在原来的线程处理完对象之前访问该对象，也就有可能会看到该对象处于损坏或无效状态。

在并发代码执行过程中，Java 的核心为防止潜在的对象破坏所提供的唯一保护就是互斥锁，它在实际的应用程序中用起来可能非常复杂。第 12 章会详细介绍 JMM 是如何工作的，还会给出使用线程和锁的一些实践。

2.6 认识不同的JVM

很多开发人员可能只熟悉 Oracle 的 Java 实现。我们已经认识了来自 Oracle 实现中的虚拟机——HotSpot。不过还有其他一些实现，本书也会进行不同程度的探讨。

OpenJDK

OpenJDK 是一个有趣的特例。它是一个开源（GPL）项目，提供了 Java 的参考实现。

该项目由 Oracle 领导和支持，并为其 Java 版本的发布提供了基础。

Oracle

Oracle 的 Java 是最广为人知的实现。它基于 OpenJDK，但在 Oracle 的专有许可证下重新授权。几乎所有对 Oracle Java 的改动都是以向 OpenJDK 公共仓库提交的方式开始的（除了尚未公开披露的安全修复之外）。

Zulu

Zulu 是一个免费的 OpenJDK 实现（基于 GPL 许可证），完全通过 Java 认证，并由 Azul Systems 公司提供。它不受专有许可证的约束，可以自由地重新分发。Azul 是为数不多的为 OpenJDK 提供付费支持的供应商之一。

IcedTea

Red Hat 是 Oracle 之外第一家能够生产完全通过认证的 Java 实现（基于 OpenJDK）的厂商。IcedTea 已通过全面认证并可以重新分发。

Zing

Zing 是一款高性能的专有 JVM。它完全通过了 Java 认证，由 Azul Systems 公司开发。它仅支持 64 位 Linux，是为配置了超大堆（TB 级）和大量 CPU 的服务器级系统设计的。

J9

IBM 的 J9 最初是一款专有的 JVM，后来开源了（就像 HotSpot 一样）。它现在建立在 Eclipse 开放运行时项目（OMR）之上，并构成 IBM 专有产品的基础。它完全兼容 Java 认证。

Avian

就认证而言，Avian 实现并不是完全兼容 Java。之所以把它放入这个列表中，是因为它是一个有趣的开源项目，对于有兴趣了解 JVM 工作细节的开发人员来说是一个很好的学习工具，而不是一个完全可用于生产的解决方案。

Android

Google 的 Android 项目有时被认为"基于 Java"。不过情况其实有点复杂。Android 最初使用的是 Java 类库的不同实现（来自净室设计的 Harmony 项目），还使用了一个交叉编译器来生成支持非 JVM 的虚拟机的不同文件格式（.dex）。

在这些实现中，本书的绝大部分内容集中在 HotSpot 上。书中的材料同样适用于 Oracle Java、Azul Zulu、Red Hat IcedTea 和所有其他基于 OpenJDK 的 JVM。

> 在做版本对比时，各种基于 HotSpot 的实现之间基本上没有性能方面的差异。

书中还有一些与 IBM J9 和 Azul Zing 相关的材料。这里只是希望读者了解这些替代方案，而不是提供一个明确的指南。有些读者可能希望更深入地探索这些技术，我们鼓励他们通过设定性能目标，然后以进行测量和比较的方式来入手。

Android 正朝着使用 OpenJDK 8 类库，并在 Android 中直接提供支持的方向发展。由于这个技术栈与其他的例子相去甚远，因此本书就不再进一步讨论 Android 了。

关于许可证的说明

几乎所有我们要讨论的 JVM 都是开源的，事实上，它们中的大多数是从基于 GPL 协议授权的 HotSpot 衍生出来的。基于 Eclipse 协议授权的 IBM 的 Open J9 和商业化的 Azul Zing（虽然 Azul 的 Zulu 产品是基于 GPL 的）是例外。

Oracle Java（从 Java 9 开始）的情况要稍微复杂一些。尽管它来自 OpenJDK 代码库，但它是专有的，**不是**开源软件。Oracle 通过让 OpenJDK 的所有贡献者都签署许可协议来实现这一点，该协议允许对他们对 OpenJDK 的 GPL 和 Oracle 的专有许可证的贡献进行双重许可。

Oracle Java 的每个更新版本都是 OpenJDK 主线之外的一个分支，以后的版本都不会在分支上进行修补。这就避免了 Oracle 和 OpenJDK 产生分歧，并解释了 Oracle JDK 和基于相同源代码的 OpenJDK 的二进制文件并没有实质差别。

这意味着 Oracle JDK 和 OpenJDK 之间唯一的真正区别是许可证。这似乎无关紧要，但 Oracle 的许可证中包含了一些开发人员应该注意的条款。

- Oracle 并不授予你在自己的组织之外重新分发其二进制文件的权利（比如，以 Docker 映像形式发布）。
- 在得到许可（通常为支持合同）之前，不得对 Oracle 的二进制文件应用二进制补丁。

Oracle 还提供了其他一些商用功能和工具，只能配合 Oracle JDK 在其许可条款内使用。不过这种情况将随着 Oracle 未来发布的 Java 版本的变化而改变，我们将在第 15 章中讨论。

在规划新的绿场部署（greenfield deployment）时，开发人员和架构师应该仔细考虑如何选择 JVM 厂商。一些大型组织，尤其是 Twitter 和阿里巴巴，甚至维护自己的 OpenJDK 私有构建版本，不过这里需要的技术投入也是许多公司无法企及的。

2.7 JVM的监控和工具

JVM 是一个成熟的执行平台，它为运行中的应用程序的注入、监控和可观测性提供了很多技术选择。用于 JVM 应用程序的这些工具的主要技术有：

- Java 管理扩展（Java management extension，JMX）
- Java 代理
- JVM 工具接口（JVM tool interface，JVMTI）
- The serviceability agent（SA）

JMX 是一种功能强大的通用技术，用于控制和监控 JVM 以及在其上运行的应用程序。它支持从客户端应用程序中以常规方式修改参数和调用方法。本书不会讨论 JMX 的方方面面。然而，JMX（以及与它相关的网络传输机制 RMI）是 JVM 管理功能的一个基本方面。

Java 代理是一个用 Java 编写的工具组件（也是因此而得名），它利用 `java.lang.instrument` 中的接口来修改方法的字节码。要安装某个代理，需要向 JVM 提供一个启动标志：

```
-javaagent:<path-to-agent-jar>=<options>
```

代理的 JAR 包中必须包含一个清单文件（manifest），并包含 `Premain-Class` 属性。此属性包含代理类的名称，该类必须实现一个公开的、静态的 `premain()` 方法，充当 Java 代理的注册钩子。

如果 Java Instrumentation API 还不够，那么可以使用 JVMTI。这是 JVM 的一个原生接口，所以使用该接口的代理必须用原生编译型语言编写——主要是 C 或 C++。我们可以将其看作一个通信接口，支持原生代理监控 JVM 事件并接收事件通知。要安装本地代理，需要的标志有点差别：

```
-agentlib:<agent-lib-name>=<options>
```

或

```
-agentpath:<path-to-agent>=<options>
```

JVMTI 代理必须用原生代码编写，这意味着更有可能写出损害正在运行的应用程序甚至导致 JVM 崩溃的代码。

在可能的情况下，通常应该选择编写 Java 代理，而不是编写 JVMTI 代码。虽然 Java 代理写起来也更容易，但有些信息是无法通过 Java API 获取的，要访问这些数据，JVMTI 可能是唯一的选择。

最后一种方法是 SA。这是一组 API 和工具，可以对外公开 Java 对象和 HotSpot 数据结构的信息。

SA 不需要在目标虚拟机中运行任何代码。相反，HotSpot SA 使用符号查找和进程内存读取等基本原语来实现调试功能。SA 可以调试活跃 Java 进程以及核心转储文件（也称为**崩溃转储文件**）。

VisualVM

除了众所周知的 `javac` 和 `java` 等二进制文件外，JDK 还提供许多其他有用的工具，其中一个经常被忽略的工具就是 VisualVM，它是一个基于 NetBeans 平台的图形化工具。

> `jvisualvm` 是早期 Java 版本中现已过时的 `jconsole` 工具的替代品。如果还在使用 `jconsole`，则应该转移到 VisualVM（有一个兼容的插件，可以让 `jconsole` 以插件形式在 VisualVM 中运行）。

最近的 Java 版本已经提供了 VisualVM 的稳定版本，而且 JDK 中的这个版本通常是够用的。但是，如果需要使用更近的版本，可以到 Java.net 网站下载最新版。下载后，必须确保将 `visualvm` 二进制文件添加到你的路径中，否则将得到 JRE 默认的版本。

从 Java 9 开始，VisualVM 将从主发行版中删除，因此开发人员必须单独下载二进制文件。

第一次启动 VisualVM 时，它会对所运行的机器进行校准，所以此时不应该运行其他应用程序，以免影响性能校准。校准完成后，VisualVM 将完成启动并显示一个闪屏。在 VisualVM 中，人们最熟悉的视图是 Monitor 界面，如图 2-4 所示。

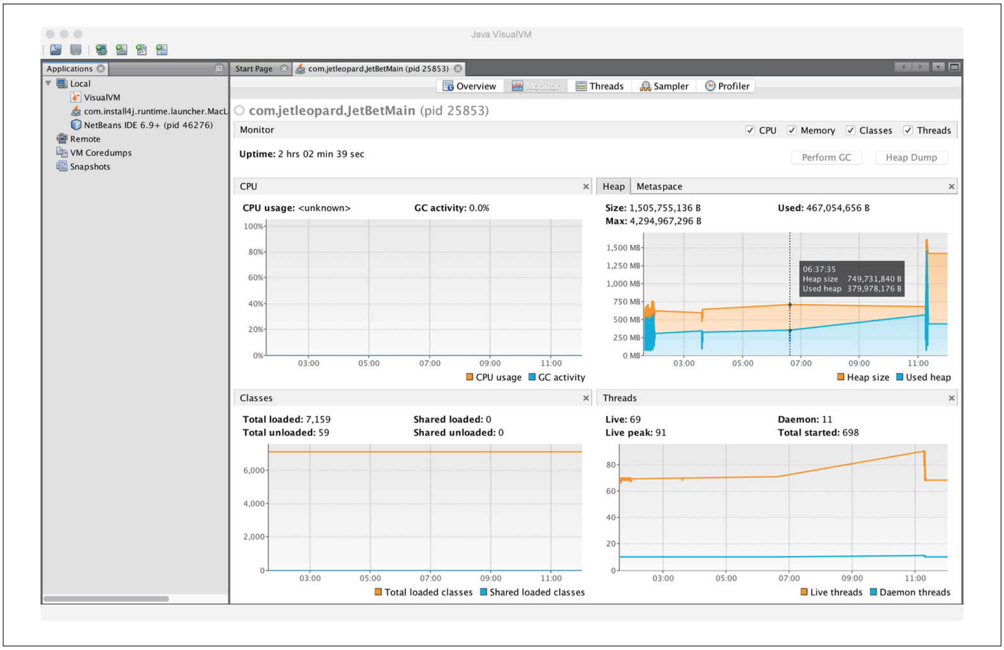


图 2-4：VisualVM 的 Monitor 界面

VisualVM 用于实时监控正在运行的进程，它使用的是 JVM 的**附加**（attach）**机制**。根据进程是本地进程还是远程进程，工作原理略有不同。

本地进程相当直接。VisualVM 会在屏幕左边列出这些进程。双击其中一个进程，右侧面板中会出现一个新的选项卡对应该进程。

要连接到远程进程，远程端必须接受入站连接（通过 JMX）。对于标准的 Java 进程，这意味着 `jstatd` 必须在远程主机上运行（详情请参阅 `jstatd` 的手册页面）。

很多应用程序服务器和执行容器直接在服务器上提供了与 `jstatd` 等效的功能。这样的进程不需要单独的 `jstatd` 进程。

要连接到一个远程进程，请输入将在选项卡上使用的主机名和显示名。默认的连接端口是1099，但是这个可以轻松更改。

VisualVM 为用户提供了 5 个开箱即用的选项卡。

概述（Overview）
: 提供有关 Java 进程的信息摘要，其中包括传入的完整标志和所有系统属性。它还会显示正在执行的、确切的 Java 版本。

监控（Monitor）
: 这个选项卡与传统的 JConsole 视图最为相似。它显示的是 JVM 的高层观测信息，包括 CPU 和堆的使用情况。它还显示加载和卸载的类的数量，以及运行的线程数量。

线程（Threads）
: 正在运行的应用程序中的每个线程都会显示在一个时间轴上，包括应用程序线程和虚拟机线程。从中我们可以看到每个线程的状态以及少量的历史记录。如果需要，还可以生成线程转储。

抽样器（Sampler）和剖析器（Profiler）
: 在这些视图中，可以访问简化的 CPU 和内存利用率抽样。第 13 章会进行更全面的讨论。

VisualVM 采用了插件架构，所以很容易向核心平台添加额外的工具以扩展核心功能，其中包括允许与 JMX 控制台交互并桥接到遗留的 JConsole 的插件，以及非常有用的垃圾收集插件 VisualGC。

2.8 小结

本章简要介绍了 JVM 的整体结构。虽然我们只能触及一些最重要的主题，但事实上，这里提到的几乎每个主题背后都有丰富完整的内容，值得进一步研究。

第 3 章将讨论操作系统和硬件工作原理的一些细节。这为 Java 性能分析人员了解观测结果提供了必要的背景。我们还将更详细地研究计时子系统，它将作为一个完整的例子来说明虚拟机和原生子系统是如何交互的。

第 3 章

硬件与操作系统

为什么 Java 开发人员应该关心硬件呢？

多年来，计算机产业一直在摩尔定律的推动下发展。该定律是英特尔创始人 Gordon Moore 就处理器性能的长期趋势提出来的一个假设。摩尔定律（其实是一个观察或推断）有多种表达方式，不过最常见的一种是：

> 批量生产的芯片上的晶体管数量大概每隔 18 个月翻一倍。

这个现象说明计算能力随时间周期呈指数级增长。摩尔定律诞生于 1965 年，它代表的是一个不可思议的长期趋势，这在人类发展史上几乎是无与伦比的。摩尔定律的影响在现代世界的许多（即便不是大多数）领域具有变革性。

> 数十年来，不断有人宣告摩尔定律已死。现在已经有非常充分的理由支持这一点，即芯片技术的这种不可思议的演进过程实际上已经（最终）走到了尽头。

按照摩尔定律，现代计算机上的“晶体管预算”不断增加，为了充分利用它们，硬件变得越来越复杂。而要充分挖掘新的能力，运行在硬件上的软件平台，其复杂性也要增加。因此，尽管软件可以利用的计算能力更强大了，但是要获得性能提升，还必须依赖硬件的复杂基础。

对于普通的应用程序开发人员来说，这种性能的巨大提升成了复杂软件的希望之花。软件应用程序现在已经遍及整个社会的各个方面。

换句话说：

> 软件正在吞噬整个世界。
>
> ——Marc Andreessen

正如我们将看到的，Java 已经成为不断增长的计算能力的受益者。这门语言及运行时的设计正好适应并利用了处理器能力提升的趋势。不过，要充分挖掘可用资源，真正追求性能的 Java 程序员仍然需要了解平台底层的原理和技术。

后续章节将探索现代 JVM 的软件架构，以及在平台和代码级别上优化 Java 应用程序的技术。在进入那些主题之前，先快速了解一下现代硬件和操作系统会有助于后续的学习。

3.1 现代硬件简介

很多关于硬件系统结构的大学课程，仍然在以一种简单且容易理解的经典方式教授硬件。这种毫无新意的讲解方式，关注的就是一个简单的基于寄存器的机器，支持算术、逻辑、加载和存储操作。因此，与 CPU 实际做什么相比，它过分强调了 C 语言编程，让人将其当作真理之源。简单来说，这是现代一种与事实不符的世界观。

自 20 世纪 90 年代以来，应用程序开发人员的世界在很大程度上是以 Intel x86/x64 架构为中心的。在这个领域中，技术已经发生了翻天覆地的变化，许多先进的特性也已成为舞台上的重要部分。关于处理器操作的那种简单的心智模型现在完全错误，而基于那种模型的简单推理，也很容易得出完全错误的结论。

为了帮助解决这个问题，本章将讨论 CPU 技术中的一些先进技术。我们将从内存的行为开始讨论，因为到目前为止，这对于一个现代 Java 开发人员而言是最重要的。

3.2 内存

随着摩尔定律的发展，晶体管数量的指数级增长最初带来的是越来越快的时钟速度。原因很明显：更快的时钟速度意味着每秒能完成更多指令。同时，处理器的速度也相应大大提升，我们如今所使用的频率在 2 GHz 以上的处理器，比第一代 IBM PC 最初的 4.77 MHz 芯片要快数百倍。

然而，不断提高的时钟速度揭示了另一个问题：更快的芯片需要更快的数据流来执行。如图 3-1 所示，[1] 随着时间的推移，主存储器已经无法满足处理器核心对新数据的需求。

注 1：此示例取自由 John L. Hennessy 和 David A. Patterson 所著的《计算机体系结构：量化研究方法（第 5 版）》一书，此书已由人民邮电出版社出版，详情请见 ituring.cn/book/888。——编者注

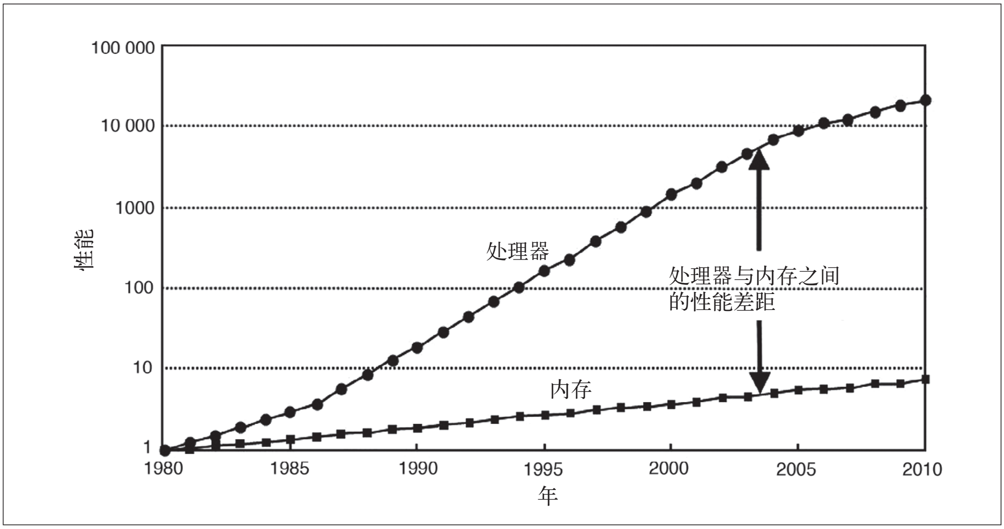


图 3-1：内存的速度和晶体管数量（Hennessy 和 Patterson，2011 年）

这就导致了一个问题：如果 CPU 在等待数据，那么再快的周期也无意义，因为 CPU 只能闲置，直到所需的数据到来。

内存高速缓存

为了解决这个问题，我们引入了 CPU 高速缓存。这些是 CPU 上的内存区域，速度比 CPU 的寄存器慢，但比主存快。其思路是，对于频繁访问的内存位置，让 CPU 将对应的内存数据的副本装填到高速缓存中，而不是每次都到内存中读取。

现代的 CPU 有几层高速缓存，其中最常访问的高速缓存位于处理核心附近。离 CPU 最近的高速缓存通常称为 L1（表示"第 1 级高速缓存"），下一层称为 L2，以此类推。不同的处理器架构，高速缓存的数量和配置也会有所不同，但通常的选择是每个执行核心都有一个专用的、私有的 L1 和 L2 高速缓存，而 L3 高速缓存是部分核心或所有核心共享的。这些高速缓存在加快访问时间方面的效果如图 3-2 所示。[2]

注 2：这里显示的访问时间以每个操作所需的时钟周期数计算，数据由 Google 提供。

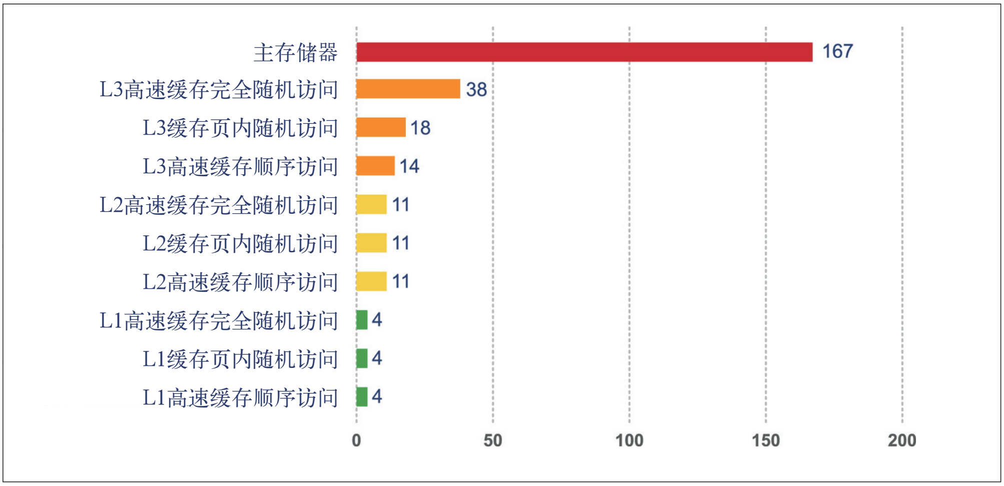


图 3-2：不同类型内存的访问时间

高速缓存架构的这种实现方法可以缩短访问时间，并有助于核心储备足够的数据用于操作。由于时钟速度与访问时间的差距，现代 CPU 上往往会有更多的晶体管预算被投入到高速缓存上。

由此产生的设计如图 3-3 所示。图中显示了 L1 和 L2 高速缓存（每个 CPU 核心私有的）和一个共享的 L3 高速缓存，L3 是这个 CPU 上的所有核心共用的。主存储器通过北桥组件进行访问，因为要穿越这个总线，所以到主存储器的访问时间大幅下降。

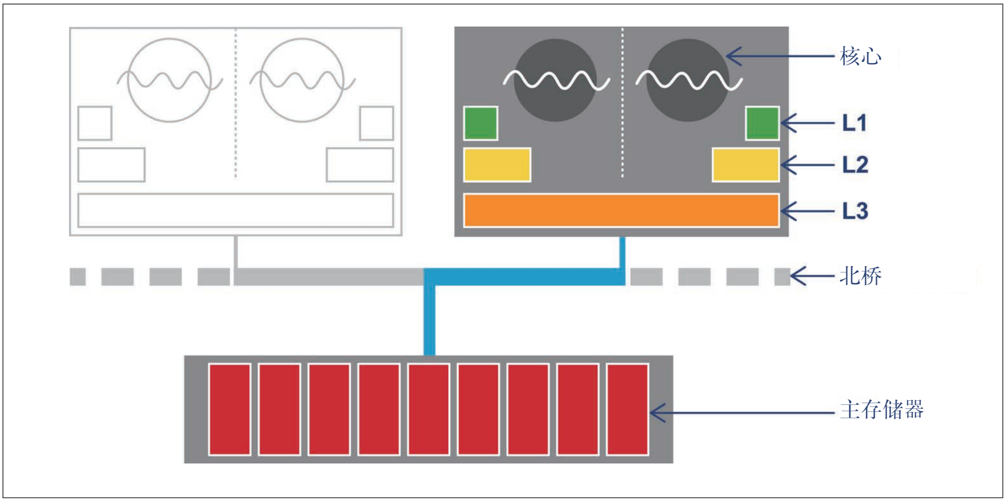
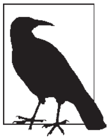


图 3-3：CPU 和内存的总体架构

虽然高速缓存架构的加入极大地改善了处理器的吞吐量，但也带来了一系列新问题。这些问题包括确定如何将数据从内存提取到高速缓存，以及如何将数据从高速缓存写回内存。这个问题的解决方案通常被称为**高速缓存一致性协议**（cache consistency protocol）。



当在并行处理环境中应用这种类型的高速缓存时，还会出现其他问题，本书后面会介绍。

在最底层，很多不同的处理器上有一种叫作 MESI（及其变种）的协议。它为高速缓存中的任何一行定义了 4 种状态，每一行（通常为 64 字节）只能处于以下状态中的一种：

- 已修改（modified），但尚未刷新到主存；
- 独占（exclusive），只出现在当前的高速缓存中，但是数据和主存一致；
- 共享（shared），可能也出现在其他高速缓存中，但是数据和主存一致；
- 无效（invalid），可能未被使用，会尽快丢弃。

该协议的理念是多个处理器可以同时处于共享状态。但是，如果一个处理器转换到其他任何一个有效状态（独占或已修改），那么这将强制所有其他处理器进入无效状态，如表 3-1 所示。

表3-1：不同处理器之间允许的MESI状态

	M	E	S	I
M	-	-	-	Y
E	-	-	-	Y
S	-	-	Y	Y
I	Y	Y	Y	Y

该协议是通过将某个处理器想要改变状态的意图广播出去来工作的。通过共享的内存总线发送一个电信号，其他处理器就能够意识到，状态转换的完整逻辑如图 3-4 所示。

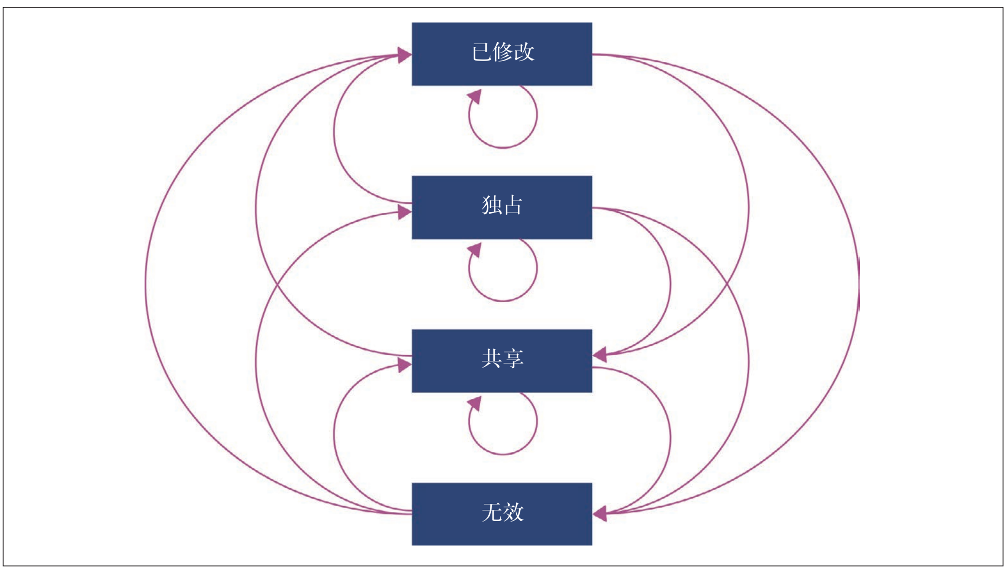


图 3-4：MESI 状态转换图

最初，处理器会直接将每个高速缓存操作写入主内存。这叫作**直写**（write-through）行为，这种方式在过去和现在都非常低效，而且需要大量的内存带宽。但更新的处理器实现了**写回**（write-back）行为，只有当已修改的（脏的）高速缓存块要被替换时，才将其写入内存，这种方式可以极大降低回到主存的流量。

高速缓存技术的总体效果是大大提高了从内存读写数据的速度，速度用内存带宽表示。**突发速率**（burst rate），或者说理论上的最大内存带宽，基于如下几个因素：

- 内存时钟频率；
- 内存总线带宽（通常为 64 位）；
- 接口数量（现代机器通常使用两个内存接口）。

如果是 DDR RAM，还要乘以 2（DDR 代表“双倍数据速率”，因为它在时钟信号的上升沿和下降沿都会进行通信）。将这个公式应用到 2015 年的商用硬件上，可以得到理论上的最大写入速度为 8~12 GB/s。当然，在实际应用中，速度可能会受到系统中许多其他因素的限制。按照实际情况，它给出了一个适度的参考值，让我们可以看看硬件和软件能够接近到什么程度。

下面写一些简单的代码来实际体验一下高速缓存硬件，如示例 3-1 所示。

示例 3-1　高速缓存示例

```
public class Caching {

    private final int ARR_SIZE = 2 * 1024 * 1024;
    private final int[] testData = new int[ARR_SIZE];

    private void run() {
        System.err.println("Start: "+ System.currentTimeMillis());
        for (int i = 0; i < 15_000; i++) {
            touchEveryLine();
            touchEveryItem();
        }
        System.err.println("Warmup finished: "+ System.currentTimeMillis());
        System.err.println("Item     Line");
        for (int i = 0; i < 100; i++) {
            long t0 = System.nanoTime();
            touchEveryLine();
            long t1 = System.nanoTime();
            touchEveryItem();
            long t2 = System.nanoTime();
            long elItem = t2 - t1;
            long elLine = t1 - t0;
            double diff = elItem - elLine;
            System.err.println(elItem + " " + elLine +" "+ (100 * diff / elLine));
        }
    }

    private void touchEveryItem() {
        for (int i = 0; i < testData.length; i++)
            testData[i]++;
    }
```

```
        private void touchEveryLine() {
            for (int i = 0; i < testData.length; i += 16)
                testData[i]++;
        }

        public static void main(String[] args) {
            Caching c = new Caching();
            c.run();
        }
    }
```

直观地说，`touchEveryItem()` 所做的工作是 `touchEveryLine()` 的 16 倍，因为它必须更新 16 倍的数据条目。然而，这个简单的例子说明了在处理 JVM 性能问题时，直觉会严重地把我们引入歧途。接下来看看 `Caching` 类的一些示例输出，如图 3-5 所示。

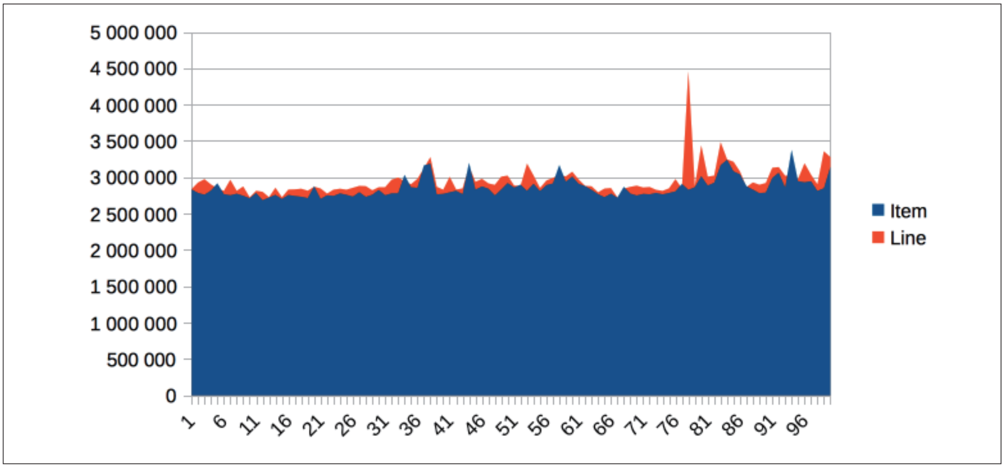


图 3-5：高速缓存示例的时间消耗情况

这张图显示了每个函数运行 100 次的情况，旨在演示几种不同的效果。首先可以注意到两个函数所花的时间非常相近，所以“16 倍的工作量”这一直观预期显然是错误的。

相反，这段代码考验的主要是内存总线，将数组的内容从主存中转移到高速缓存中，供 `touchEveryItem()` 和 `touchEveryLine()` 来处理。

从数字的统计结果来看，虽然结果还算比较一致，但存在个别异常值与中位数相差 30%~35%。

总的来说，可以看到简单内存函数的每一次迭代大约需要 3 毫秒（平均 2.86 毫秒）来遍历 100 MB 的内存块，这样计算出的有效内存带宽略低于 3.5 GB/s。这比理论上的最大值要小，但仍然是一个合理的数字。

现代 CPU 有一个硬件预取器（prefetcher），它可以发现数据访问中的可预测模式（通常就是通过一个常规的“步幅”来访问数据）。本例利用这一事实，来使内存访问带宽接近实际的最大值。

Java 性能的关键主题之一是应用程序对对象分配率的敏感性。后面我们还会多次讨论这个问题，这个简单的示例提供了一个基本的标准，让我们了解分配率可能会上升到多高。

3.3 现代处理器特性

硬件工程师有时会把摩尔定律所带来的新特性称为“花掉晶体管预算”。内存高速缓存最明显的地方就是使用的晶体管越来越多了，但这些年也出现了其他技术。

3.3.1 翻译后备缓冲器

还有一个非常重要的用途是用于另一种高速缓存中，即翻译后备缓冲器（translation lookaside buffer，TLB）。它充当的是页表的高速缓存，而页表的作用是将虚拟内存地址映射到物理地址。 TLB 可以大大提升访问虚拟地址下的物理地址这一非常频繁的操作速度。

如果没有 TLB，即使页表被保存在 L1 高速缓存中，所有的虚拟地址查找也要花费 16 个时钟周期。这样的性能是无法接受的，所以 TLB 基本上对所有现代芯片都是必备的。

3.3.2 分支预测和推测执行

现代处理器上出现的高级处理器技巧之一是分支预测（branch prediction），用于防止出现处理器必须等待计算条件分支所需要的值的情况。现代处理器有多阶段的指令流水线，这意味着一个 CPU 周期的执行会被分解成若干个独立的阶段。处理器可以同时执行多条指令（它们处于不同的执行阶段）。

在这个模型中，条件分支是不确定的，因为在条件被计算完成之前，并不知道分支之后的下一条指令是什么。这可能会导致处理器停滞（stall）若干个时钟周期（实际最多达 20 个），因为它实际上要清空分支后面的多阶段流水线。

众所周知，推测性执行是导致 2018 年年初发生的影响大量 CPU 重大安全问题的原因。

为了避免这种情况，处理器用晶体管构建了一个启发式策略，以判断更有可能采用哪个分支。基于这种猜测，CPU 用赌博的方式填充流水线。如果判断成功，那么 CPU 就会像往常一样继续执行。如果判断错误，那么部分被执行的指令就要被丢弃，CPU 还必须接受清空流水线的惩罚。

3.3.3 硬件存储器模型

在多核系统中，有一个关于内存的必须回答的核心问题是，多个不同的 CPU 如何能够一致地访问同一个内存位置。

这个问题的答案高度依赖于硬件，一般来说，`javac`、JIT 编译器和 CPU 都允许改变代码的执行顺序，但前提是任何改变都不能影响当前线程所观察到的结果。

比如，假设有这样一段代码：

```
myInt = otherInt;
intChanged = true;
```

这两条赋值语句之间没有其他代码，所以执行线程不需要关心它们以什么顺序发生，因而 Java 执行环境可以随意改变这两条指令的顺序。

然而，这可能意味着在另一个能看到这些数据项的线程中，顺序可能会发生变化，而另一个线程读取到的 `myInt` 的值也可能是旧值，尽管 `intChanged` 看上去是 `true`。

这种类型的重排序（store 被重排序到 store 之后）在 x86 芯片上是不可能发生的，但正如表 3-2 所示，在其他架构中，这种情况是有可能发生的，而且确实发生了。

表3-2：硬件存储器的支持

	ARMv7	POWER	SPARC	x86	AMD64	zSeries
load 被重排序到 load 之后	Y	Y	-	-	-	-
load 被重排序到 store 之后	Y	Y	-	-	-	-
store 被重排序到 store 之后	Y	Y	-	-	-	-
store 被重排序到 load 之后	Y	Y	Y	Y	Y	Y
原子操作和 load 重排序	Y	Y	-	-	-	-
原子操作和 store 重排序	Y	Y	-	-	-	-
是否存在不一致的指令高速缓存	Y	Y	Y	Y	-	Y

在 Java 环境中，Java 内存模型（JMM）被明确地设计为一种弱模型，以考虑到不同处理器类型之间内存访问的一致性的差异。要确保多线程代码正常工作，主要是靠正确使用锁和易失性（volatile）访问。第 12 章中还会再次讨论这个重要的主题。

近年来出现了一种趋势，即要获得更好的性能，软件开发人员就要更好地了解硬件的工作原理。Martin Thompson 等人引入了一个术语“机械共鸣”（mechanical sympathy）来描述这种方法，特别是当应用于低延迟和高性能领域的问题时。在最近关于无锁的算法和数据结构的研究中可以看到它的身影，本书最后将对此进行讨论。

3.4 操作系统

操作系统的意义在于控制对资源的访问，这些资源必须在多个执行进程之间共享。因为所有的资源都是有限的，而所有的进程都很贪婪，所以必须有一个中心系统来对访问做仲裁和计量。在这些稀缺资源中，最重要的两个通常为内存和 CPU 时间。

通过内存管理单元（MMU）及其页表的虚拟寻址是实现内存访问控制的关键特性，它可以防止一个进程破坏另一个进程所拥有的内存区域。

本章前面提到过 TLB 的一种硬件特性，即它可以改善物理内存的查找时间。这种缓冲区的使用改善了软件对内存的访问时间，提高了性能。但是，MMU 的级别通常很低，以致开发人员无法直接影响或意识到。相反，让我们仔细看看操作系统进程调度器，因为它控制着对 CPU 的访问，并且是操作系统内核中用户更能看到的部分。

3.4.1 调度器

对 CPU 的访问是由进程调度器控制的。它使用了一个所谓的**运行队列**，作为有资格运行但必须等待 CPU 的线程或进程的等待区。在现代系统上，因为想要运行的线程 / 进程实际上总是比能够运行的多，所以需要一种机制来解决 CPU 争用问题。

调度器的工作是响应中断，并管理对 CPU 核心的访问。一个 Java 线程的生命周期如图 3-6 所示。理论上，Java 规范允许使用这样的线程模型，而 Java 线程也未必对应于操作系统线程。然而，在实践中，这种“绿色线程”方法并没有被证明确实有用，所以在主流的操作环境中已经被放弃了。

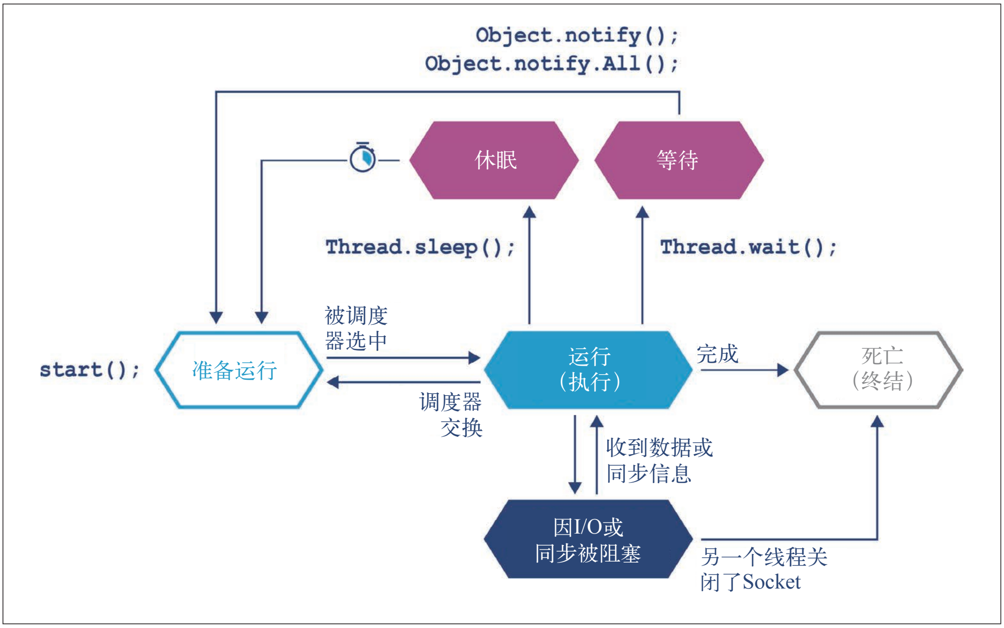


图 3-6：线程生命周期

在这个相对简单的视图中，系统配置的是单核处理器，操作系统的调度器会将线程移动到单核上和将线程从单核上移走。在时间片结束时（在比较老的操作系统中通常是 10 毫秒或 100 毫秒），调度器会将线程移动到运行队列的后面，以等待线程到达队列的前面并有资格再次运行。

如果一个线程想主动放弃其时间片，有两种方式，一种是指定固定的时间（通过 `sleep()`），一种是等待某个条件被满足（使用 `wait()`）。最后线程也可能阻塞在 I/O 或软件锁上。

初次使用此模型时，考虑仅具有一个执行核心的机器可能会有帮助。当然，真正的硬件会更复杂，几乎任何现代机器都有多个处理器核心，从而可以真正同时执行多个执行路径。这意味着推测真正的多处理环境下的执行情况是非常复杂且违反直觉的。

操作系统的一个经常被忽视的特点是，由于其本身的性质，它们会引入一段时间，应用程序的代码在这段时间内是没有在 CPU 上运行的。一个已经完成了其时间片的进程，直到它再次到达运行队列的前端时，才会回到 CPU 上。又因为 CPU 是一种稀缺资源，所以代码等待的时间往往要比运行的时间长。

这也意味着对于实际想要观测的进程，所要生成的统计数据会受到系统中其他进程行为的影响。这种“抖动”和调度开销是导致观测结果出现噪声的主要原因。第 5 章中将讨论实际结果的统计特性和处理方法。

查看调度器的动作和行为的一个最简单方法就是尝试观察操作系统为实现调度而带来的开销。下面的代码执行了 2000 个单独的 2 毫秒休眠。每一个休眠都会涉及线程被送到运行队列的尾部，然后必须等待一个新的时间片。所以，这段代码的总耗费时间可以让我们对典型进程的调度开销有一定的了解：

```
long start = System.currentTimeMillis();
for (int i = 0; i < 2_000; i++) {
    Thread.sleep(2);
}
long end = System.currentTimeMillis();
System.out.println("Millis elapsed: " + (end - start) / 4000.0);
```

根据操作系统的不同，在不同的地方运行这段代码会导致截然不同的结果。大多数 Unix 系统所报告的开销大约在 10%~20%。早期版本 Windows 的调度器很差，比如在某些 Windows XP 版本上，所报告的调度开销高达 180%（因此，1000 个 1 毫秒的睡眠需要 2.8 秒）。甚至有报道称，一些专有操作系统的厂商在其发布的系统中插入了部分代码，以检测运行的是否为基准测试，如果是的话则骗过这些指标。

计时（timing）对于性能测量、进程调度以及应用栈中的许多其他部分至关重要，所以让我们来看看 Java 平台是如何处理计时的，并深入了解 JVM 和底层操作系统是如何提供支持的。

3.4.2 时间问题

尽管存在像 POSIX 这样的工业标准，但是不同的操作系统还是会有非常不同的行为。比如，考虑 `os:::javaTimeMillis()` 函数。在 OpenJDK 中，这个函数会包含与具体操作系统相关的调用，而正是这些调用在完成实际的工作，并最终提供 Java 的 `System.currentTimeMillis()` 方法要返回的值。

正如 2.5 节所述，由于它依赖于必须由主机操作系统提供的功能，因此 `os::javaTimeMillis()` 必须实现为原生方法。下面是在 BSD Unix（比如 Apple 的 macOS 操作系统）上实现的函数：

```
jlong os::javaTimeMillis() {
  timeval time;
  int status = gettimeofday(&time, NULL);
  assert(status != -1, "bsd error");
  return jlong(time.tv_sec) * 1000 + jlong(time.tv_usec / 1000);
}
```

Solaris、Linux，甚至 AIX 的版本都与 BSD 的情况非常相似，但 Microsoft Windows 版本的代码完全不同：

```
jlong os::javaTimeMillis() {
  if (UseFakeTimers) {
    return fake_time++;
  } else {
    FILETIME wt;
    GetSystemTimeAsFileTime(&wt);
    return windows_to_java_time(wt);
  }
}
```

Windows 使用了一个 64 位的 `FILETIME` 类型来存储自 1601 年以来的、以 100 纳秒为单位的时间，而不是 Unix 的 `timeval` 结构。Windows 还有一个系统时钟的“真实精度”的概念，具体取决于可用的物理计时硬件。因此，在不同的 Windows 机器上，Java 的计时调用的行为会有很大差异。

操作系统之间的差异并不只是计时，下一节将看到更多内容。

3.4.3 上下文切换

上下文切换是指操作系统调度器移除当前正在运行的线程或任务，并将其替换为等待中的线程。有几种不同类型的上下文切换，但一般来说，它们都涉及交换正在执行的指令和线程的栈状态。

无论是用户线程之间的上下文切换，还是从用户模式进入内核模式（有时称为模式切换），都是一个成本很高的操作。后一种情况尤为重要，因为用户线程在其时间片中途可能需要切换到内核模式以便执行某些功能。但是，这种切换将强制清空指令和其他高速缓存，因为用户空间代码所访问的内存区域与内核空间通常没有任何交叉。

上下文切换到内核模式，将使 TLB 和其他可能的高速缓存失效。当调用返回时，这些缓存必须重新填充，因此内核模式切换的影响会一直持续到控制权返回到用户空间之后。这将导致系统调用的真实成本被掩盖，如图 3-7 所示。[3]

注 3：Livio Stoares and Michael Stumm,“FlexSC:Flexible System Call Scheduling with Exception-Less System Calls，”in OSDI’10，*Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*（Berkeley，CA：USENIX Association，2010）, 33–46。

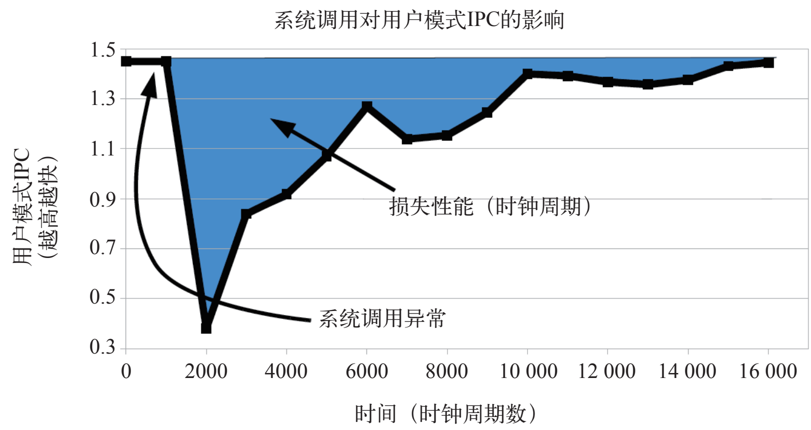


图 3-7：系统调用的影响（Soares 和 Stumm，2010 年）

为了尽可能地缓解这种情况，Linux 提供了一种称为 vDSO（virtual dynamically shared object）的机制。这是用户空间中的一个内存区域，用于加速那些并不是真的需要内核权限的系统调用。它通过避免实际执行上下文切换到内核模式来提升速度。下面通过一个示例来看看它在实际的系统调用中是如何工作的。

一个非常常见的 Unix 系统调用是 `gettimeofday()`。它返回的是操作系统所理解的“挂钟时间”（wallclock time）。在幕后，它实际上只是读取一个内核数据结构来获取系统时钟时间。由于该操作无副作用，因此它实际上不需要特权访问。

如果可以用 vDSO 来安排这个数据结构，并将其映射到用户进程的地址空间中，就不需要执行切换到内核模式的操作。因此，也就不用再付出图 3-7 中所示的重新填充成本了。

考虑到大多数 Java 应用程序需要访问计时数据，所以如果能提升其性能，那肯定很受欢迎。通过 vDSO 机制将这个例子稍作推广，即使只能用在 Linux 上，也是有用的技术。

3.5 一个简单的系统模型

本节将介绍一个简单的模型，用于描述可能出现的性能问题的基本来源。这个模型通过可以在基本子系统上观测到的操作系统信息来表示，并可以直接关联到标准的 Unix 命令行工具的输出。

该模型基于一个简单的概念：一个 Java 应用程序运行在一个 Unix 或类 Unix 的操作系统上。图 3-8 显示了该模型的基本组件，其中包括：

- 应用程序运行所基于的硬件和操作系统；
- 应用程序运行所在的 JVM（或容器）；
- 应用程序代码本身；

- 应用程序调用的任何外部系统；
- 来到该应用程序的外部输入请求流量。

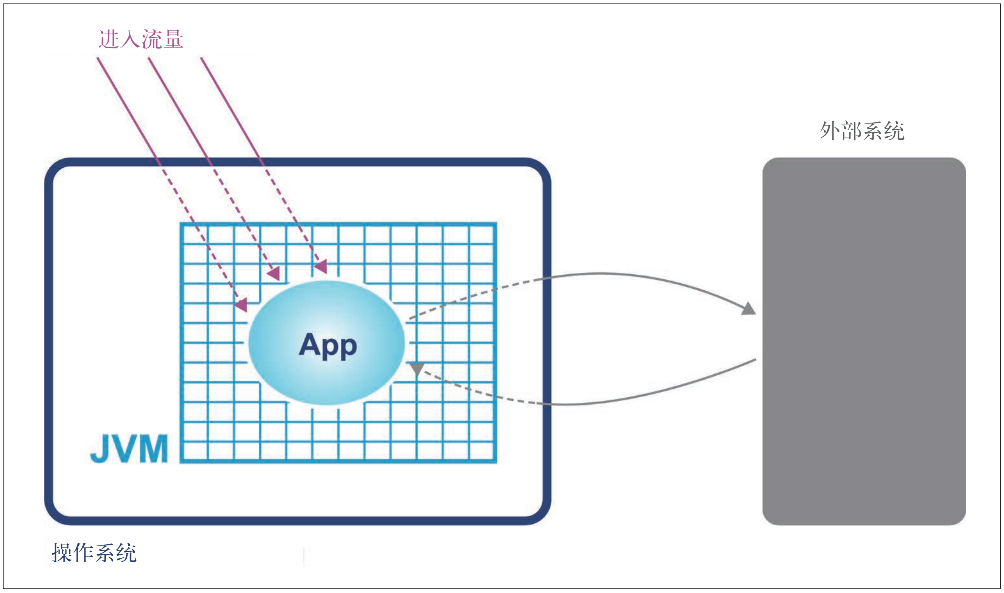


图 3-8：简单系统模型

系统中的任何一方面都可能导致某个性能问题。有一些简单的诊断技术可以用来缩小或隔离系统的特定部分，从而定位到出现性能问题的潜在原因，下一节将介绍。

3.6 基本探测策略

所谓性能良好的应用程序，一个定义就是能有效地利用系统资源，这包括 CPU 利用率、内存、网络或 I/O 带宽。

> 如果一个应用程序导致一个或多个资源的使用达到极限，其结果就是出现了性能问题。

任何性能诊断的第一步就是要识别出哪种资源的使用达到了极限。如果不处理资源短缺问题，就无法调优适当的性能指标——要么增加可用资源，要么提高使用效率。

另外值得注意的是，操作系统本身通常不应该成为影响系统利用率的主要因素，其作用是代表用户进程管理资源，而不是自己去消耗资源。这一规则唯一例外的情况是当资源已经非常稀缺时，操作系统就会难以分配足够的资源来满足用户的需求。对于大多数现代的服务器级硬件来说，这种情况只有在 I/O（偶尔是内存）的需求大大超过供应能力时才会出现。

3.6.1 利用CPU

应用程序性能的一个关键指标是 CPU 的利用率。CPU 周期往往是应用程序所需要的最关键资源，因此，有效地利用这些资源对于获得良好的性能至关重要。在高负载时期，应用程序的目标应该是尽可能地接近 100% 的利用率。

> 当分析应用程序性能时，系统必须在足够的负载下才能发挥其性能。空闲的应用程序的行为通常对性能工作没有任何意义。

每个性能工程师都应该知道的两个基本工具是 `vmstat` 和 `iostat`。在 Linux 和其他 Unix 系统上，这些命令行工具分别提供了对虚拟内存和 I/O 子系统的当前状态的直接且非常有用的信息。虽然这些工具只提供了整个主机层面的数字，但往往足以为更详细的诊断方法指明方向。接下来看一个如何使用 `vmstat` 的示例：

```
$ vmstat 1
 r  b swpd  free    buff  cache   si   so  bi  bo   in   cs us sy  id wa st
 2  0    0 759860 248412 2572248    0    0   0  80   63  127  8  0  92  0  0
 2  0    0 759002 248412 2572248    0    0   0   0   55  103 12  0  88  0  0
 1  0    0 758854 248412 2572248    0    0   0  80   57  116  5  1  94  0  0
 3  0    0 758604 248412 2572248    0    0   0  14   65  142 10  0  90  0  0
 2  0    0 758932 248412 2572248    0    0   0  96   52  100  8  0  92  0  0
 2  0    0 759860 248412 2572248    0    0   0   0   60  112  3  0  97  0  0
```

`vmstat` 后面的参数 `1` 表示希望它提供持续的输出（直到通过 Ctrl-C 中断），而不是只提供一个快照。新的输出行每秒打印一次，这使得性能工程师可以在执行初始性能测试的同时让这个输出保持运行（也可以将其输出到一个日志中）。

`vmstat` 的输出相对容易理解，它包含了大量的有用信息，分为以下几个部分。

1. 前两列显示了可运行的（`r`）和阻塞的（`b`）进程的数量。
2. 内存部分显示了交换区和空闲内存的数量，以及用于缓冲区和高速缓存的内存数量。
3. 交换区部分显示了从磁盘上交换进来的内存量，以及交换到磁盘上的内存量（`si` 和 `so`）。现代服务器级的机器通常不会有太多的交换活动。
4. 块的输入输出计数（`bi` 和 `bo`）显示的是从块（I/O）设备接收到的 512 字节大小的块的个数，以及发送给块（I/O）设备的个数。
5. 在系统部分，显示的是中断次数（`in`）和每秒上下文切换次数（`cs`）。
6. CPU 部分包含一些直接相关的指标，以百分比表示 CPU 时间。它们依次是用户时间（`us`）、内核时间（`sy`，表示“系统时间”）、空闲时间（`id`）、等待时间（`wa`）和“被盗时间”（`st`，用于虚拟机）。

本书剩余部分还会介绍很多其他更复杂的工具。不过千万不要忽视我们所掌握的基本工具。复杂的工具往往会误导我们的行为，而那些简单的工具离进程和操作系统比较近，反而可以清晰地给出系统的实际表现。

让我们来考虑一个例子。3.4.3 节讨论了上下文切换的影响，在图 3-7 中，我们也看到了

从完整上下文切换到内核空间的潜在影响。然而，无论是在用户线程之间还是在内核空间中，上下文切换都不可避免地会浪费 CPU 资源。

一个调优好的程序应该能最大限度地利用其资源，尤其是 CPU。对于主要依赖计算的工作负载（“CPU 密集型”问题），其目标是使在用户空间工作的 CPU 利用率接近 100%。

换句话说，如果观察到 CPU 的利用率没有接近 100% 的用户时间，那么接下来显然要问的就是：为什么不呢？是什么原因导致程序没有实现这个目标？是由锁引起的意外上下文切换导致的问题吗？还是由于 I/O 争用导致的阻塞？

`vmstat` 工具可以在大多数操作系统（尤其是 Linux）上显示发生上下文切换的次数，所以运行 `vmstat 1` 可以让分析人员看到上下文切换的实时效果。如果一个进程没有实现 100% 的用户空间 CPU 利用率，而且还显示出很高的上下文切换率，那么这个进程很可能是阻塞在 I/O 上了，或是遇到了线程锁的争用。

然而单靠 `vmstat` 的输出还不足以完全分辨出到底是哪种情况。`vmstat` 可以帮助分析人员检测到 I/O 问题，因为它提供了一个粗略的 I/O 操作视图，但是要实时检测线程锁争用情况，应该使用 VisualVM 这样的工具来显示运行进程中的线程状态。另外一个常用的工具是基于统计的线程剖析器，它可以对栈进行采样以提供阻塞代码的视图。

3.6.2 垃圾收集

我们在第 6 章将看到在 HotSpot JVM（迄今为止最常用的 JVM）中，内存是启动时分配，并在用户空间内进行管理。这意味着不需要使用系统调用（如 `sbrk()`）来分配内存，进而也说明了用于垃圾收集的内核切换活动相当少。

因此，如果某个系统表现出的系统 CPU 利用率过高，那么它一定没有把大量时间花在垃圾收集上，因为垃圾收集活动会让用户空间的 CPU 周期飙升，但不会影响内核空间的 CPU 利用率。

另外，如果一个 JVM 进程在用户空间使用了 100%（或接近这个值）的 CPU，那么垃圾收集往往就是罪魁祸首。在分析一个性能问题时，如果简单的工具（如 `vmstat`）显示出 CPU 利用率一直在 100%，但几乎所有的周期都是在用户空间消耗的，那我们就应该会问，这样的利用率是 JVM 还是用户代码导致的？几乎在所有情况下，JVM 的用户空间 CPU 利用率高都是由垃圾收集子系统造成的，所以一个有用的经验法则是检查垃圾收集日志并查看日志中新条目增加的频率。

JVM 中的垃圾收集日志成本非常低，以致即使用最精确的整体成本测量手段也无法可靠地将其与随机背景噪声区分开来。将垃圾收集日志用作分析的数据源也是非常有价值的。因此，对于所有的 JVM 进程，特别是生产环境中的 JVM，启用垃圾收集日志非常有必要。

本书后面将对垃圾收集以及垃圾收集日志进行大量讨论。但是现在，我们鼓励读者咨询他们的运维人员并确认生产环境中是否开启了垃圾收集日志。如果没有，那么第 7 章的一个核心要点就是建立一个策略来实现这一点。

3.6.3 I/O

文件 I/O 在传统上一直是整个系统性能中不太好解决的方面之一。这在一定程度上是因为它与五花八门的物理硬件关系更密切，工程师们经常用“旋转的铁锈”之类的话来调侃它，但这也是因为 I/O 不像操作系统的其他部分那样有清晰的抽象。

以内存为例，作为一个隔离机制，虚拟内存非常优雅，而且表现很好。然而，I/O 没有一个同等的抽象来为应用程序开发人员提供适当的隔离。

幸运的是，虽然大多数 Java 程序会涉及一些简单的 I/O，但是大量使用 I/O 子系统的应用程序相对较少，特别是大多数程序不会在大量消耗 CPU 或内存的同时，再试图让 I/O 也饱和起来。

不仅如此，惯用的运维实践也带来了这样一种文化：产品工程师已经意识到 I/O 的限制，他们会积极地监控进程是不是在大量使用 I/O。

对于性能分析员 / 工程师来说，只要对应用程序的 I/O 行为有所了解就足够了。像 `iostat`（甚至是 `vmstat`）这样的工具都有基本的计数器（比如，输入或输出的块），这往往是进行基本诊断所需要的全部内容，特别是当我们可以假设每个主机上只有一个 I/O 密集型应用程序时，这些工具的价值更大。

最后，关于 I/O，值得一提的是，既依赖 I/O，又对性能有严格要求的这类应用程序越来越多了。

内核旁路I/O

对于某些高性能应用程序来说，使用内核将数据从比如网卡上的缓冲区复制到用户空间区域的成本过高。相反，可以使用专用的硬件和软件将数据直接从网卡上映射到用户可访问的区域。这种方法既避免了“双重复制”，也避免了跨越用户空间和内核之间的边界，正如图 3-9 所示。

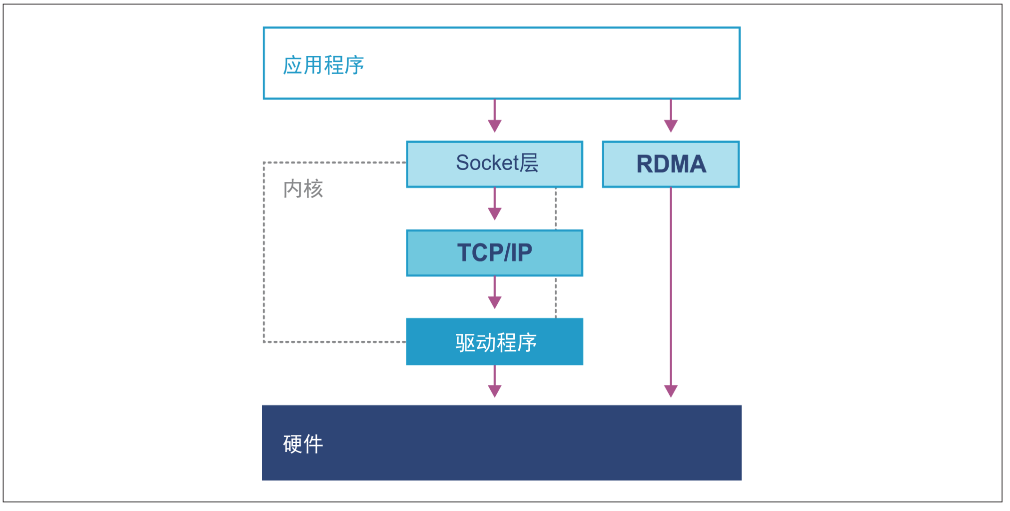


图 3-9：内核旁路 I/O

然而，Java 并没有为这种模型提供具体的支持，而是希望使用它的应用程序依赖于定制（原生）的库来实现所需语义。这种模式非常有用，越来越多需要高性能 I/O 的系统都实现了它。

> 在某种程度上，这让人联想到 Java 的 New I/O (NIO) API，它的引入就是为了支持 Java 的 I/O 绕过 Java 堆直接与原生内存和底层 I/O 一起工作。

本章讨论了运行在“裸金属”（bare metal）上的操作系统。然而，现在有越来越多的系统在虚拟化环境中运行，因此，本章最后来简单看看虚拟化如何从根本上改变了我们对 Java 应用程序性能的看法。

3.6.4　机械共鸣

机械共鸣是这样一种理念，充分理解硬件并抱以欣赏对于获得额外的性能非常有利。

> 要成为一名赛车手，你不必成为工程师，但是一定要有机械共鸣。
>
> ——Jackie Stewart

这个词是由 Martin Thompson 引入到计算机领域的，直接引用了 Jackie Stewart 关于车的说法。然而除了极端情况之外，在处理生产问题并研究提高应用程序的整体性能时，对本章所概括的一些关注点有个基本的了解也是很有帮助的。

对于很多 Java 开发人员来说，虽然机械共鸣很重要，但是它有可能被忽视。这是因为，为让开发人员从各种性能问题中解脱出来，JVM 提供了一个远离硬件的抽象层。通过理解 JVM 以及它与硬件之间的交互关系，开发人员也可以成功地将 Java 和 JVM 应用于高性能、低延迟领域。还有很重要的一点需要注意，即 JVM 实际上会让推理解决性能和机械共鸣问题变得更加困难，因为有更多需要考虑的问题。第 14 章将描述高性能日志和消息传递系统是如何工作的，以及如何提升机械共鸣。

来看一个例子：高速缓存行的行为。

本章讨论了处理器高速缓存的好处。我们可以通过高速缓存行来提取内存块。在多线程环境中，当有两个线程试图读取或写入位于同一缓存行上的某个变量时，缓存行可能会引发问题。

当两个线程现在试图修改同一高速缓存行时，就会发生竞争。第一个线程将使第二个线程上的高速缓存行失效，进而导致它需要从内存中重新读取。而一旦第二个线程执行了修改操作，又会使第一个线程的缓存行失效。这种互相影响的行为会导致性能下降，我们称其为**伪共享**（false sharing），那么如何解决这种问题呢？

机械共鸣告诉我们，首先需要理解正在发生的事情，只有理解了之后才能决定如何解决。在 Java 中，对象中的字段布局是没有保证的，这意味着很容易出现多个变量共享同一缓存行的情况。规避这个问题的一个方法是在变量周围添加填充（padding），强制它们进入不同的高速缓存行上。在 14.3.1 节讲到“队列”（queue）时，我们会看到，它使用了 Agrona 项目中的一个低延迟队列来解决这个问题。

3.7 虚拟化

虚拟化有多种形式，但最常见的一种形式是在已运行的操作系统上以单个进程的形式运行一个操作系统的副本。这导致了图 3-10 所示的情况，虚拟环境作为一个进程运行在一个非虚拟化的（或者说“真正的”）操作系统内，而这个非虚拟化的系统是在真正的硬件之上执行的。

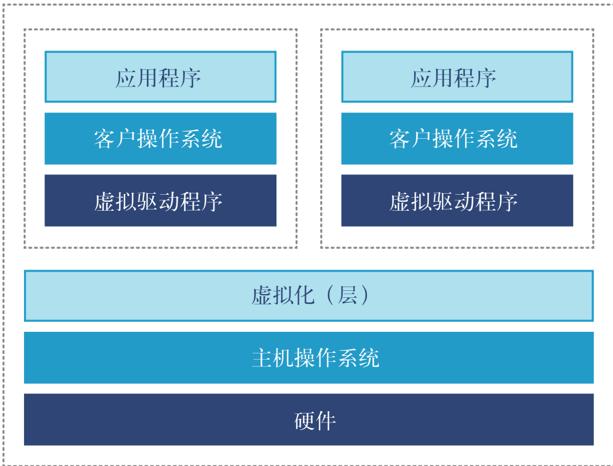


图 3-10：操作系统的虚拟化

如果要全面讨论虚拟化、相关理论及其对应用程序性能的影响，那就离题万里了。不过提几点虚拟化引起的差异似乎是恰当的，特别是考虑到当前运行在虚拟化环境或云环境中的应用程序越来越多了。

尽管虚拟化早在 20 世纪 70 年代就在 IBM 大型机环境中开发出来了，但直到最近，x86 架构才有能力支持“真正的”虚拟化，这一点可以用如下 3 个条件来判断：

- 在虚拟化操作系统上运行的程序，其行为应该与在裸机（即非虚拟化）上运行时基本相同；
- 虚拟机监视器（hypervisor）必须调解所有对硬件资源的访问；
- 虚拟化的开销必须尽可能小，不能占用太多的执行时间。

在一个正常的非虚拟化系统中，操作系统内核以特殊的特权模式运行（因此需要切换到内核模式），这使得操作系统可以直接访问硬件。但是，在虚拟化系统中，禁止客户操作系统（Guest OS）直接访问硬件。

一种常见的方法是将特权指令改写为非特权指令。此外，操作系统内核的一些数据结构需要被“影子化”（shadowed），以防止在上下文切换过程中出现过度的高速缓存冲刷（如 TLB）。

某些现代的 Intel 兼容 CPU 设计了一些硬件特性，旨在提高虚拟化操作系统的性能。但是很明显，即使有硬件辅助，在虚拟环境中运行也会让性能分析和调优变得复杂。

3.8 JVM和操作系统

通过支持 Java 代码的公用接口，JVM 提供了一个独立于操作系统的可移植执行环境。然而，对于某些基础服务，比如线程调度（甚至像非常常见的获取系统时钟的时间）就必须要访问底层的操作系统。

这种能力是通过本地方法提供的，此类方法用关键字 `native` 来标记。它们用 C 语言编写，但是可以像普通 Java 方法一样访问。这类接口被称为 Java 本地接口（Java native interface，JNI）。比如，`java.lang.Object` 声明了下面这些非私有的本地方法：

```
public final native Class<?> getClass();
public native int hashCode();
protected native Object clone() throws CloneNotSupportedException;
public final native void notify();
public final native void notifyAll();
public final native void wait(long timeout) throws InterruptedException;
```

所以这些方法处理的都是相对底层的平台问题。来看一个更直接和更熟悉的例子：获取系统时间。

考虑 `os::javaTimeMillis()` 函数。这是负责实现 Java 中 `System.currentTimeMillis()` 静态方法的、特定于系统的代码。虽然负责实际工作的代码是用 C++ 实现的，但在 Java 中通过一个 C 语言的“桥接”代码访问。来看一下这个代码在 HotSpot 中实际是如何调用的。

如图 3-11 所示，本地 `System.currentTimeMillis()` 方法被映射到了 JVM 入口点方法 `JVM_CurrentTimeMillis()` 中。映射通过包含在 java/lang/System.c 文件中的 Java 本地接口 `Java_java_lang_System_registerNatives()` 来实现。

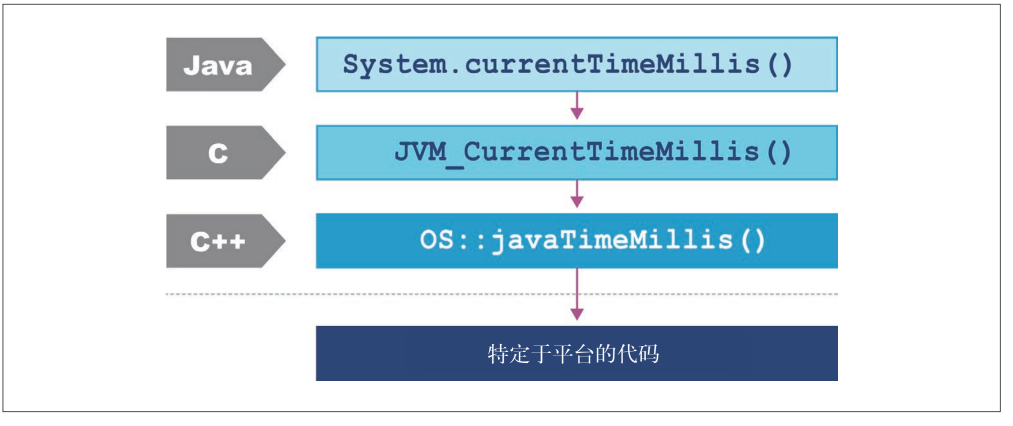


图 3-11：HotSpot 调用栈

`JVM_CurrentTimeMillis()` 是对虚拟机入口点方法的调用。它虽然表现为一个 C 函数，但实际是由 C++ 实现的，然后通过 C 调用约定导出，最终会调用通过一系列 OpenJDK 宏包装起来的 `os::javaTimeMillis()`。

该方法是在 `os` 命名空间中被定义的，不出所料，它取决于操作系统。在 OpenJDK 中，方法的定义由源代码中与操作系统相关的子目录提供。这里简单演示了 Java 独立于平台的部分是如何调用到底层操作系统和硬件提供的服务中的。

3.9 小结

在过去的 20 年里，处理器设计和现代硬件发生了巨大的变化。在摩尔定律和工程上的限制（特别是内存速度相对较慢）的驱动下，处理器设计的进步已经变得让人有点难以理解。缓存未命中率已经成为衡量一个应用程序性能最明显的领先指标。

在 Java 领域，JVM 的设计允许它使用额外的处理器核心，甚至对于单线程应用程序代码也是如此。这意味着与其他环境相比，Java 应用程序已经从硬件趋势中获得了明显的性能优势。

随着摩尔定律的消逝，人们的注意力再次转向软件的相对性能上。注重性能的工程师至少需要了解现代硬件和操作系统的基本要点，以确保他们能够充分利用硬件，而不是反其道而行之。

下一章将介绍性能测试的核心方法论，并且讨论性能测试的主要类型、需要承担的任务以及性能工作的整个生命周期。我们还将列举一些在性能分析领域常见的最佳实践和反模式。

第 4 章

性能测试模式与反模式

性能测试会因不同的原因而进行。本章将介绍团队可能希望执行的不同类型的测试，并讨论每种类型的最佳实践。

本章的后半部分将概述一些可能会困扰性能测试或团队的常见反模式，并阐释如何重构的解决方案，以防止它们成为团队的问题。

4.1　性能测试的类型

性能测试经常因为错误的原因而实施，或者实施得很糟糕。虽然发生的原因各不相同，但其根源往往在于没有理解性能分析的本质，认为“有总比没有好”，正如我们将反复看到的，这种看法真假参半，非常危险。

比较常见的错误之一是笼统地谈论“性能测试”而不涉及具体内容。事实上，在一个系统上可以执行多种不同类型的大规模性能测试。

> 好的性能测试是量化的。它们提出的问题可以得到一个数字化的答案，用于作为实验输出来处理并进行统计分析。

本书将讨论不同类型的性能测试，它们的目标大部分情况下是独立的，偶尔会有交叉，所以在思考任何一个特定的测试时，你都应该小心谨慎。在规划性能测试时，一个很好的经验法则就是直接写下测试要回答的可量化的问题（并向管理层 / 客户确认），并说明这些问题对于被测试的应用程序很重要的原因。

以下是一些最常见的测试类型，对于每种类型，我们还给出了一个示例问题。

延迟测试

　终端到终端的交易时间是多少？

吞吐量测试

　目前的系统容量能处理多少个并发交易？

负载测试

　系统是否能处理某个特定的负载？

压力测试

　系统的临界点是什么？

耐久性测试

　系统长期运行时会发现哪些性能异常现象？

容量规划测试

　当增加额外的资源时，系统的规模是否能按预期扩展？

退化测试

　当系统发生部分故障时会出现什么情况呢？

下面让我们依次来详细了解这些测试类型。

4.1.1　延迟测试

这是最常见的性能测试类型之一，因为它通常与管理层直接感兴趣的系统观测对象密切相关：我们的客户等待交易或页面加载的时间有多长？这是一把双刃剑，因为延迟测试所要回答的量化问题似乎非常明显，以致它可能会掩盖识别其他类型的性能测试所需要的量化问题的必要性。

> 对延迟进行调优，其目标通常是直接改善用户体验，或者满足服务等级协议（SLA）。

然而，即使在最简单的情况下，延迟测试也有一些必须谨慎对待的微妙之处，其中最值得注意的是，对于衡量应用程序对请求的响应程度而言，简单的平均数并不是很有用（相关内容参见 5.3 节）。

4.1.2　吞吐量测试

吞吐量可能是性能测试时第二个常见的观测量。在某些意义上，甚至可以将其看作等同于延迟。

例如，当进行延迟测试时，有一点非常重要，即在生成延迟结果的分布时，需要说明和控制正在进行的并发交易数。

所观测到的系统延迟应该在已知和受控的吞吐量水平下给出。

同样，我们通常在监控延迟的同时进行吞吐量测试，通过留意延迟分布何时会突然发生变化——实际上是系统的一个“临界点”（也称为**拐点**）来判断“最大吞吐量”。正如我们将看到的，压力测试的目的就是确定这样的点以及它们出现时的负载水平。

然而，吞吐量测试是测量系统开始降级之前观测到的最大吞吐量。

4.1.3　负载测试

负载测试与吞吐量测试（或压力测试）的不同之处在于，它通常被定义为二进制测试，即系统能不能处理这个预设的负载。在预期的业务事件之前，有时要实施负载测试，比如新的客户或市场有可能使应用程序的流量激增。还有一些情况下也需要进行此类测试，比如广告活动、社交媒体事件和“病毒式营销”。

4.1.4　压力测试

我们可以把压力测试看作一种确定系统还有多少余量空间的方法。该测试通常是通过将系统置于稳定的交易状态来进行的，也就是在某个指定的吞吐量水平（通常是当前的峰值）之下。

压力测试会缓慢地增加并发交易数，直到我们观测的系统数据开始下降。

通过观测数据刚开始下降时的值，就可以确定系统在吞吐量测试中可以达到的最大吞吐量。

4.1.5　耐久性测试

有些问题只有在更长的时间内（通常以天为单位）才会出现。这些问题包括缓慢的内存泄漏、高速缓存污染和内存碎片化，尤其是对于使用并发标记和扫描垃圾收集器（CMS）的应用程序来说，它们最终可能会出现并发模式失败。相关内容参见 7.3 节。

为了检测这些问题，通常的方法是进行耐久性测试（也称为浸泡测试）。这些测试在平均（或高）利用率下运行，但是系统的负载要处于观测之下。测试期间要密切监控资源利用水平，以发现任何故障或资源耗尽的情况。

这种类型的测试在快速响应或低延迟的系统中非常常见，因为这些系统通常无法承受一次 Full GC 周期所造成的 STW 事件的时间（参见第 6 章和后续章节，以了解更多与 STW 事件和相关垃圾收集概念有关的信息）。

4.1.6　容量规划测试

容量规划测试与压力测试有许多相似之处，但它们是两种不同类型的测试。压力测试的作

用是找出当前系统能够承受的负载，而容量规划测试更具前瞻性，旨在确定系统在升级后能够承受的负载。

因此，容量规划测试通常作为常规计划的一部分进行，而非应对特定事件或威胁。

4.1.7 退化测试

退化测试也称为部分失效测试。虽然关于弹性和失败恢复测试的一般性讨论超出了本书的范围，但是可以这样说，在监管和审查最严密的环境（包括银行和金融机构）中，失败恢复测试不仅会极为严肃地进行，而且通常都会有细致深入的规划。

就目的而言，这里唯一要考虑的弹性测试类型是退化测试。该测试的基本方法是，当系统在相当于通常生产量级的模拟负载下运行时，一个组件或整个子系统突然失去了能力，此时观察系统会有何表现。例如，应用程序服务器集群突然丢失机器、数据库突然丢失 RAID 磁盘，或网络带宽突然下降等。

在进行退化测试时，主要观测对象包括交易延迟分布和吞吐量。

部分故障测试有一个特别有意思的子类型，叫作**混沌猴**（chaos monkey）。这是以 Netflix 公司的一个项目命名的，该项目是为了验证其基础设施的健壮性。

混沌猴背后的理念是，在一个真正的弹性架构中，单个组件的故障不应该导致级联故障，也不应该对整个系统产生实际影响。

混沌猴试图通过随机终止生产环境中实际使用的活跃进程来演示这一点。

为了成功地实现混沌猴类型的系统，公司必须具有最高水平的系统卫生、服务设计和运维优势。然而，这也是越来越多的公司和团队所关注和向往的领域。

4.2 最佳实践入门

在决定性能调优的工作重点时，如下 3 个黄金法则可以提供有用的指导：

- 确定你所关心的是什么，并想好如何去衡量它；
- 优化重要的东西，而不是容易优化的东西；
- 先抓住要点。

第二点有一个相反的说法，就是提醒自己不要陷入这样的陷阱：过于重视容易测量到的量。并不是每一个可观测到的量对业务都有重要意义，但有时人们会倾向于报告一个容易测量的量，而不是一个正确的量。

4.2.1 自上而下的性能测试

谈到 Java 性能，很多工程师一开始容易忽略的一个方面就是，对 Java 应用程序进行大规模的基准测试通常要比试图获得小段代码的精确性能数据容易。第 5 章将详细讨论这个问题。

从整个应用程序的性能行为入手，这种方法通常称为**自上而下**的性能测试。

为了充分利用自上而下的方法，测试团队需要一个测试环境，并清楚地理解需要测量和优化的内容，以及性能工作如何融入整个软件开发生命周期。

4.2.2 创建一个测试环境

建立一个测试环境是大多数性能测试团队的首要任务之一。只要有可能，这个环境应该是对生产环境的所有方面的精确复制，不仅包括应用服务器（哪些服务器应该有相同数量的CPU、相同的操作系统和 Java 运行时版本等），还包括 Web 服务器、数据库、负载均衡器和网络防火墙等。任何服务（例如不容易复制的第三方网络服务，或者没有足够的 QA 能力来处理与生产环境等同的负载）都需要模拟，以创建一个有代表性的性能测试环境。

有时，团队会尝试复用或通过分时共享的方式来使用现有的 QA 环境以进行性能测试。对于较小的环境或一次性测试来说这是可能的，但不要低估管理成本以及它可能带来的调度和协调问题。

如果性能测试环境与它所要代表的生产环境明显不同，那就往往无法得到任何对生产环境有意义的东西或预测能力。

对于传统的不是基于云的环境，与生产环境类似的性能测试环境相对容易实现，因为团队只需购买和生产环境中使用的同等数量的物理机器，然后以与生产环境完全相同的方式来配置。

管理层有时会抵制这些额外的基础设施成本。这几乎总是得不偿失的，但令人遗憾的是，很多公司无法正确评估服务中断带来的损失，从而导致人们产生一种想法，即认为不去花钱打造一个精确的性能测试环境，所节省的成本是有意义的，这是因为他们并没有正确地考虑到 QA 环境不能正确反映生产环境所带来的风险。

技术的最新发展，特别是云技术的出现，改变了这种相当传统的局面。按需供给和自动伸缩的基础设施意味着越来越多的现代架构不再符合“购买服务器、绘制网络图以及在硬件上部署软件”的模式。将服务器基础设施视为“牲畜”（livestock）而非“宠物”（pet）[1] 的 DevOps 方法 ，意味着更为动态化的基础设施管理方法正在普及。

这使得构建一个像生产环境的性能测试环境更具挑战性。然而，这也让创建一个在不用时就能关闭的测试环境成为可能。虽然它可以为项目节约大量成本，但需要一个按计划启动和关闭环境的适当的流程。

注 1：原文是“livestock, not pets”，也有人用“pets vs cattle”，指的是两种服务器管理模式。宠物模式，就是把服务器当作自己的宠物，每台服务器都有一个可爱的名字，当服务器出现问题时，要加以修复。牲畜模式，就是不再特殊对待服务器，给它们起像 Web01、Web02、Web03 这样的名字，就像在牲畜的耳朵上挂的编号。如果服务器出现问题，就直接用另一个服务器将其替换掉。——译者注

4.2.3 确定性能要求

回顾一下 3.5 节介绍的简单系统模型，该模型清楚地表明了系统的整体性能并不完全取决于应用程序代码，容器、操作系统和硬件都有其作用。

因此，作为评估性能的指标，不应该仅从代码的角度考虑，还必须将系统作为一个整体，考虑对客户和管理层来说都很重要的观测量。这些通常被称为性能**非功能性需求**（nonfunctional requirement，NFR），是我们想要优化的关键指标。

有些目标是显而易见的：

- 将 95% 的交易时间减少 100 毫秒；
- 改进系统，使现有硬件上的吞吐量提高 5 倍；
- 将平均响应时间减少 30%。

其他目标可能不那么明显：

- 将服务普通客户的资源成本降低 50%；
- 确保即使应用程序集群减少了 50%，系统仍然能够支持 25% 的响应目标；
- 将每 25 毫秒延迟下的客户下降率减少 25%。

与干系人进行公开讨论，以精确确定应该测量的内容以及要达到的目标是至关重要的。理想情况下，此讨论应该成为性能优化第一次启动会议的一部分。

4.2.4 Java特有的问题

性能分析的大部分科学知识适用于任何现代软件系统。然而，由于 JVM 的特性，性能工程师应该注意并仔细考虑一些额外问题。该问题主要来自 JVM 的动态自我管理能力，例如内存区域的动态调优。

Java 特有的一个非常重要的问题与 JIT 编译有关。现代 JVM 会分析哪些方法正在运行，以选定可以进行 JIT 编译的候选方法，并为其生成优化的机器代码。这意味着如果一个方法没有被 JIT 编译，那它会处于如下两种情况之一：

- 该方法运行频率不够高，无法被编译；
- 该方法太大或太复杂，无法进行编译分析。

第二种情况更为少见。然而，对于基于 JVM 的应用程序，刚开始的性能优化手段是打开简单的日志，看看哪些方法被编译了，并确保在应用程序的关键代码路径上的重要方法都会被编译。

第 9 章将详细讨论 JIT 编译，我们还会演示一些简单的技术，并说明如何确保应用程序的重要方法能成为 JVM 进行 JIT 编译的目标。

4.2.5 将性能测试当作软件开发生命周期的一部分

有些公司和团队喜欢把性能测试看作偶尔的、一次性的活动。然而，有经验的团队往往会持续进行性能测试，特别是会把性能回归测试当作软件开发生命周期（SDLC）中不可或缺的一部分。

这需要开发人员和基础架构团队之间的协作，以控制在任何特定时间内性能测试环境中要出现的代码版本。如果没有专门的测试环境，那么这几乎是不可能实现的。

在讨论了一些最常见的性能测试的最佳实践之后，现在让我们把注意力转到团队可能陷入的陷阱和反模式上。

4.3 性能反模式

反模式是指我们在软件项目或团队中不希望出现，但在大量项目中又可以看到的行为。[2] 反模式的频繁出现，使得人们有了这样的结论或怀疑：某些潜在因素要为产生了我们不希望出现的行为负责。有些反模式乍一看似乎是合理的，其不好的方面并不明显。有些则是不良的项目实践随着时间的推移慢慢累积的结果。

在某些情况下，这些行为可能是由于社会或团队的限制、常见管理技术的滥用或者只是人类（开发人员）的本性导致的。通过对这些不希望出现的特性进行分类和归类，我们就可以开发出一种“模式语言”来讨论它们，并希望将其从项目中消除。

应该始终将性能调优视为一个非常客观的过程，并在计划阶段的早期就设定精确的目标。但是这说起来容易做起来难，因为当一个团队面临压力时，或者在合理的情况下都不能正常运作时，很容易就半途而废了。

很多读者遇到过这样的情况：在一个新的客户端将要上线，或者一个新的特性就要发布时意外出现了服务中断。如果幸运的话，中断会出现在用户验收测试（user acceptance testing，UAT）阶段；但不幸的是，它经常出现在生产环境中。这时团队就要手忙脚乱地寻找引发瓶颈的问题并加以修复。导致这样的结果通常是因为没有进行性能测试，或者团队的“专家”假设了某个条件，但是这个条件消失了。

与遵循了良好的性能测试实践并进行了开放、理性的对话的团队相比，以这种方式工作的团队更容易成为反模式的牺牲品。就像许多开发问题一样，往往是人为因素，比如沟通问题，而不是任何技术方面的原因导致应用程序出现问题。

Carey Flichel 在一篇名为“Why Developers Keep Making Bad Technology Choices”的博客文章中提供了一种有趣的分类，文中特别指出了导致开发人员做出错误选择的 5 大原因，接下来我们依次看一下。

4.3.1 厌倦

大多数开发人员有过在某个角色中感到厌倦的经历，对一些人来说，这种情况并不会持续很长时间，因为他们会在公司或其他地方寻求新的挑战或角色。但是，公司里可能没有其他的机会，换个地方可能也没有。

很多读者可能遇到过这样的开发人员，他们能够克服困难，甚至可能积极寻求更轻松的生活。然而，感到厌倦的开发人员可能会以多种方式给项目带来损害。例如，他们可能

注 2：由 William J. Brown、Raphael C. Malvo、Hays W. McCormick III 和 Thomas J. Malbray 合著的《反模式：危机中软件、架构和项目的重构》一书普及了这一术语，此书已由人民邮电出版社出版。

会引入不必要的代码复杂性，比如直接在代码中写一个排序算法，而这本来使用简单的 `Collections.sort()` 就够了。他们也可能会使用未知的或不适合当前场景的技术来构建组件，只是希望有个机会用这些技术，以此来解决他们的厌倦感。这又会导致出现下面的问题。

4.3.2 填充简历

偶尔，技术的过度使用并不是因为厌倦，而是因为开发人员想找到一个机会，在简历上增加使用某项技术的经验。在这种情况下，开发人员会积极尝试提高自己的潜在薪资和市场竞争力，因为他们即将重新进入就业市场。在一个运作良好的团队内部，很多人不太可能靠这个侥幸成功，不过项目确实可能会因为这个原因而进入不必要的路径。

由于开发人员的厌倦，或为了填充简历而增加了一项不必要的技术，其影响可能是深远而持久的，在原来的开发人员另谋高就后还会持续数年。

4.3.3 同侪压力

在做出选择时，如果关注点没有得到表达或讨论，那所做出的技术决策往往是最糟糕的。问题有可能通过几种方式表现出来。比如，也许某个初级开发人员不想在团队中的高级成员面前犯错（“负担症候群”），或者某个开发人员担心遇到对某个特定主题不了解的情况。另一种特别有害的同侪压力来自存在竞争关系的团队，大家都希望表现出更高的开发速度，因此会在没有充分探讨所有后果的情况下就仓促做出关键决定。

4.3.4 缺乏理解

因为没有了解当前工具的全部功能，所以开发人员可能会寻求引入新的工具来帮助解决问题。使用适合执行某个具体任务的新的、令人兴奋的技术组件往往很有诱惑力。然而，引入更多技术复杂性也必须考虑与当前工具的实际功能相平衡。

例如，Hibernate 有时被看作简化领域对象和数据库之间转换的解决方案。如果团队对 Hibernate 的理解相当有限，那么开发人员可能会基于在另一个项目中看过的 Hibernate 的使用情形而对其适用性做出假设。

缺乏理解会导致 Hibernate 的用法过度复杂，甚至导致无法恢复的生产环境中断。相比之下，使用简单的 JDBC 调用来重写整个数据层，可以让开发人员停留在熟悉的领域中。本书的一位作者曾经讲授过一门 Hibernate 课程，课上有位代表就遇到了完全一样的情况。他试图学习足够多的 Hibernate 知识，看看是否能让应用程序从中断中恢复过来，但最终不得不用一个周末的时间把 Hibernate 去掉了，这可不是值得羡慕的事。

4.3.5 被错误理解的问题/不存在的问题

开发人员可能经常会使用某种技术来解决某个特定的问题，但是对问题空间本身并没有进行充分研究。如果没有测量获得的性能值，那就几乎不可能理解特定解决方案是否成功。通常，整理这些性能指标有助于更好地理解问题。

为了避免出现反模式，要确保关于技术问题的交流对团队的所有参与者公开，并鼓励大家

踊跃参与，这非常重要。在事情不清楚的地方，收集事实证据和研究原型可以帮助指导团队做出决策。一项技术可能看起来很有吸引力，但是如果原型不符合要求，那么团队也可以做出更明智的决策。

4.4 性能反模式目录

本节将介绍一个简短的性能反模式目录。这份清单绝不是详尽无遗的，无疑还有更多的反模式有待发现。

4.4.1 被热门技术分心

1. 描述

最新或最酷的技术往往是第一个要调优的目标，因为探索较新技术的工作原理可能比挖掘遗留代码更令人兴奋，也有可能是伴随新技术而来的代码写得更好，更容易维护。这两个事实都促使开发人员关注应用程序中较新的组件。

2. 评论示例

> “这是初期故障，我们需要弄清真相。”

3. 现实

- 这通常只是瞎猜，而不是对应用程序进行有针对性的调优或测量。
- 开发人员可能还没有完全理解新技术，并且只会胡乱修改，而不是检查文档——这实际上会导致出现其他问题。
- 在使用新技术的情况下，网上的例子往往是针对小数据集或样本数据集的，并没有讨论如何扩展到企业规模的良好实践。

4. 讨论

在新成立的或经验不足的团队中，这种反模式很常见。他们急于证明自己，或者是为了避免被他们所认为的**遗留系统**所束缚。他们往往是较新的、“较热”的技术的拥护者，而巧合的是，这些技术恰恰是可以在任何新职位上带来薪资提升的技术。

因此，合乎逻辑的下意识结论是，任何性能问题都应该先看一看所采用的新技术。毕竟，它没有被正确地理解，所以换一种角度来看会有帮助，对吗？

5. 解决方法

- 通过测量来确定瓶颈的真正位置。
- 确保围绕新组件有足够的日志记录。
- 不能只看简化的演示，还要看最佳实践。
- 确保团队了解新技术，并在整个团队中建立最佳的实践标准。

4.4.2 被简单分心

1. 描述

团队首先瞄准的是系统中最简单的部分，而不是对应用程序进行整体剖析，客观地寻找其

中的痛点。系统中可能有些部分会被认为“相当专业”，只有最初编写它们的人才能编辑。

2. 评论示例

“先从我们理解的部分入手吧。”

“这部分是 John 写的，他正在休假。还是等他回来再看看性能吧。”

3. 现实

- 原始开发人员理解如何调优系统中他们编写的那部分（也可能是只了解自己写的那部分）。
- 对于不同的系统组件，由于没有进行知识共享或结对编程，从而导致形成了单一领域的专家。

4. 讨论

“被简单分心”和“被热门技术分心”是对称出现的，这种反模式通常出现在一个比较成熟的团队中，团队可能更习惯于维护或维持运转的角色。如果应用程序最近融入了新技术，或者要与新技术对接，那他们可能会感到畏惧，或者不想与新系统打交道。

在这种情况下，开发人员可能更愿意只剖析系统中他们熟悉的部分，希望能够在不走出其舒适区的情况下达到预期目标。

特别值得注意的是，我们刚介绍的这两种反模式都是对未知事物的反应。在“被热门技术分心”这一反模式中，这表现为开发人员（或团队）渴求学习更多知识并获得优势——本质上是一种进取策略。相比之下，“被简单分心”则是一种防御性的反应，它表现为开发人员（或团队）喜欢熟悉的事物，而不想接触具有潜在威胁的新技术。

5. 解决方法

- 通过测量来确定瓶颈的真正位置。
- 如果问题出现在一个不熟悉的组件中，就向领域专家寻求帮助。
- 确保开发人员了解系统的所有组件。

4.4.3　性能调优天才

1. 描述

管理层相信好莱坞电影中独行天才般的黑客形象，所以雇用了某个符合这种刻板印象的人，希望他能凭借其过人的性能调优技能，在公司里来回走几圈就可以解决所有的性能问题。



确实有真正的性能调优专家和公司，但是他们中的大部分人会同意任何问题都少不了测量和研究。不太可能会有一套方案能适合某种特定技术的所有应用情况。

2. 评论示例

“我肯定知道问题出在哪里了……”

3. 现实

- 被认为是魔法师或超级英雄的人唯一可能会做的事情就是穿些奇装异服。

4. 讨论

对于团队中那些认为自己还没有好到可以解决性能问题的开发人员而言，这种反模式会导致他们产生疏离感。这很令人担忧，因为在很多情况下，只需要他们进行一小部分剖析制导优化就可以带来良好的性能提升（参见第 13 章）。

这并不是说没有专家可以在特定技术上提供帮助，但认为有一个独行天才从一开始就能了解所有性能问题，这种想法很荒谬。许多作为性能专家的技术人员本身也是擅长测量并基于这些测量来解决问题的专家。

如果团队中有超级英雄类型的专家，但是他们不愿意分享知识，或不愿意分享他们为解决某个特定问题所采用的方法，那么认为他们可以解决任何性能问题可能会适得其反。

5. 解决方法

- 通过测量来确定瓶颈的真正位置。
- 确保团队中雇用的任何专家都愿意分享并成为团队的一分子。

4.4.4 按照坊间传说调优

1. 描述

当某个团队成员急于为生产中的性能问题找到解决方案时，他在网上发现了一个“神奇”的配置参数。在没有测试这个参数的情况下，团队将其应用到生产中，因为它一定会带来改进，网上就是这么说的……

2. 评论示例

> “我在 Stack Overflow 上发现了这些很有用的技巧。这改变了**一切**。”

3. 现实

- 因为开发人员并不了解这个性能技巧的上下文和依据，所以真正的影响是未知的。
- 它可能在那个特定的系统中起了作用，但这并不意味着这种改变在另一个系统中也会有好处。在现实中，它可能会让事情变得更糟。

4. 讨论

所谓性能技巧，就是某个已知问题的解决方案（或为其寻找解决方案）。性能技巧也有其“保质期”，通常会随着时间的推移而变差。比如可能某个解决方案就使现在使用的技巧在后来的软件或平台版本中失效了。

糟糕的性能建议的一个来源是管理手册，其中包含的一般性建议往往缺乏上下文。如果厂商被起诉，那律师经常会坚持用这种模糊的建议和“推荐的配置”作为辩护的额外防线。

Java 的性能与上下文有关，其中有大量的影响因素。如果把这个上下文剥离掉，那么由于执行环境的复杂性，我们就几乎无法对剩下的东西进行研究了。

> Java 平台也在不断地演进，这意味着在一个 Java 版本中提供的性能解决方案的参数，在另一个版本中可能会无效。

例如，用于控制垃圾收集算法的开关在不同版本之间经常发生变化。在旧版本（Java 7 或 Java 6）的虚拟机中可以工作的东西可能在当前版本（Java 8）中无法使用。甚至有一些在 Java 7 中合法且有效的开关，在即将到来的 Java 9 中会导致虚拟机无法启动。

配置可能只是更改了一两个字符，但如果不仔细管理，则会对生产环境产生重大影响。

5. 解决方法

- 对于会给系统的最重要方面造成直接影响的技术，只应用其中经过良好测试并充分理解的技术。
- 在用户验收测试阶段寻找和尝试参数，但是对于任何修改，重要的是证明并描述其好处。
- 与其他开发人员、运维人员或 DevOps 团队一起审查和讨论配置。

4.4.5 把责任归咎给驴

1. 描写

某些组件总是会被当成问题所在，即使它们与问题毫无关系。

例如，本书的作者之一曾在用户验收测试阶段遇到一次大规模中断故障，而此时正是上线的前一天。代码中的某个路径导致中心数据库表的一个表被锁定了。也就是说，在代码中发生了一个错误，锁没有得到释放，而是被保留了下来，使得应用程序的其他部分无法使用，直到完全重启。因为数据访问层用的是 Hibernate，所以大家马上把问题归咎于 Hibernate 了。然而在这个案例中，问题的罪魁祸首并非 Hibernate，而是一个用于处理超时异常的空 `catch` 块在这里没有清理数据库连接。开发人员花了一整天的时间才停止责怪 Hibernate，然后真正地审视自己的代码，并最终找到了真正的 bug。

2. 评论示例

> *“总是 JMS/Hibernate/A_N_OTHER_LIB 的问题。”*

3. 现实

- 没有进行充分的分析就得出这一结论。
- 经常被怀疑的对象成为调查的唯一目标。
- 团队不愿意进行更大范围的研究来确定真正的原因。

4. 讨论

这种反模式往往会通过管理人员或业务人员表现出来，因为在很多情况下，他们对技术栈并没有充分的了解，而且存在认知偏差，所以会按照模式匹配的思路来工作。不过技术人员也未能幸免于此。

如果对经常被归咎的代码库或类库之外的其他库知之甚少，那技术人员往往也会成为这一反模式的牺牲品。与进行新的调查研究相比，指出在应用程序中经常出现问题的那部分往往更容易。这可能是团队疲惫的一个信号，因为手头有很多生产问题。

Hibernate 就是一个很好的例子。在很多情况下，Hibernate 可能没有被正确设置或使用。但是，因为在过去看过该技术曾执行失败或无法执行，所以团队就会倾向于把问题归咎于它。然而，问题也很容易出现在底层查询、不恰当索引的使用、与数据库的物理连接、对象映射层或其他方面。因此通过剖析来找出确切原因是至关重要的。

5. 解决方法

- 抵制住急于得出结论的压力。
- 像往常那样执行分析。
- 向所有干系人公布分析结果（鼓励更精确地了解问题的原因）。

4.4.6 忽略大局

1. 描述

团队沉迷于尝试对应用程序的较小部分进行修改和剖析，但是没有充分认识到它们带来的整体影响。也许是看到了一个例子，或是参考了公司的其他应用程序，工程师就开始对 JVM 开关微加调整，试图获得更好的性能。

团队也可能会使用微基准测试来剖析应用程序的较小部分（要想进行正确的微基准测试相当困难，第 5 章将探讨）。

2. 评论示例

> "如果只修改这些设置，会有更好的性能。"
>
> "如果能加快方法分派时间……"

3. 现实

- 团队没有充分理解修改带来的影响。
- 团队还没有在新的 JVM 设置下对应用程序进行全面剖析。
- 微基准测试对整体系统的影响尚不确定。

4. 讨论

JVM 有数百个开关。虽然这提供了一个高度可配置的运行时，但也诱导人们去充分利用所有配置能力。这一般是错误的，因为默认值和自我管理能力通常已经足够了。有些开关还会以意想不到的方式相互组合，这使得盲目地修改更加危险。即使是同一公司的应用程序，也可能会以完全不同的方式进行运维和配置，所以花时间尝试一下推荐的设置是很重要的。

性能调优是一项统计学活动，它依赖于非常具体的执行上下文。这意味着与较小的系统相比，较大的系统通常更容易进行基准测试，因为对于较大的系统来说，大数定律有助于工程师校正平台对个体事件的扭曲。

相比之下，越是专注于系统的单一方面，就越难把构成平台的复杂环境的独立子系统（如多线程、垃圾收集、调度以及 JIT 编译）拆解出来（至少在 Java/C# 的情况下是这样）。这是极难做到的，并且处理数据统计非常敏感，往往不是软件工程师们在此过程中所能掌握的技能。因此很容易产生这样的状况，即工程师认为虽然他们正在对系统的某个方面做基准测试，但是产生的数字不能精确表示系统的行为。

再加上人类的偏见，可能会看到一些根本不存在的模式。我们会看到这样的景象，即一个性能工程师被错误的统计或差劲的控制诱导了，他强烈坚持某个性能基准测试或效果，但是他的同事无法复现。

这里还有几点需要注意。首先，如果没有一个完全模拟生产环境的用户验收测试环境，就很难评估优化的效果。其次，如果某个优化只有在应用程序面对高度压力的情况下才有效果，在一般情况下反而会伤害性能，那它是没有意义的——但是要想获得在一般情况下非常典型、又能在负载之下提供有意义的测试结果的数据集非常困难。

5. 解决方法

在对线上的开关进行任何改动之前，需要做以下事情。

- 在生产中进行测量。
- 在用户验收测试环境中每次改动一个开关。
- 确保用户验收测试环境与生产环境具有相同的压力点。
- 确保有能够代表生产系统中的正常负载的测试数据。
- 在用户验收测试环境中测试变化。
- 在用户验收测试环境中重新测试。
- 找人重新检查你的推理。
- 和他们一起讨论你的结论。

4.4.7　用户验收测试环境就是我的计算机

1. 描述

用户验收测试环境往往与生产环境有很大的不同，但是我们对这种不同并不总是能够预测和完全理解。开发人员有过在低功率的台式计算机上为高功率的生产服务器编写代码的经验。不过，开发人员用的计算机比生产中部署的小型服务器强大的这种情形也变得越来越常见了。拥有低功率的生产环境通常很容易，因为可以通过虚拟化的方式让每个开发人员都有一个这样的环境。但拥有高功率的生产环境就比较困难了，与开发人员的机器相比，它们通常拥有更多的核心、内存和高效的 I/O。

2. 评论示例

> “和生产环境完全一致的用户验收测试环境成本太高了。”

3. 现实

- 环境差异造成的服务中断所带来的损失几乎总是比买几台机器的成本大。

4. 讨论

“用户验收测试环境就是我的计算机”，这种反模式源于一种与我们之前所见并不相同的认知偏差。这种偏差坚信有个用户验收测试环境总比完全没有好。遗憾的是，所抱的这种希望完全没有理解企业环境的复杂性。要想做出任何有意义的推断，用户验收测试环境必须和生产环境类似。

在现代的自适应环境中，运行时子系统会充分利用现有资源。如果这些资源与目标部署环境大相径庭，那么运行时子系统会在不同情况下做出不同的决策，这就使我们对其怀有希望的推理全无用处了。

5. 解决方法

- 追踪服务中断带来的损失以及与客户流失相关的机会成本。

- 购买一个与生产环境相同的用户验收测试环境。
- 在大多数情况下，前者的成本远超后者，有时管理者需要做出正确的选择。

4.4.8 类似生产环境的数据很难表示

1. 描述

这种反模式也称为 DataLite 反模式，它与人们在试图表示类似于生产环境的数据时遇到的一些常见陷阱有关。以一家大型银行的交易处理系统为例，它负责处理已被预订但尚需结算的期货和期权交易。这样的系统通常一天会处理数百万条信息。现在考虑以下用户验收测试策略以及潜在的问题。

- 为了便于测试，我们的机制是从一天的消息中选择一小部分，然后将其放到用户验收测试系统中运行。

  这种方法无法捕捉到系统可能会遇到的类似突发行为。它也可能无法捕捉到在另一个交易市场开盘前，在某个特定市场上进行更多期货交易所造成的预热。

- 为了使场景更容易测试，交易和期权被更新为只使用简单的值进行断言判断。

  这并没有提供具有“真实性”的生产数据。假设我们正在使用某个外部库或系统进行期权定价，要想借助测试数据集来确定这种生产依赖是否会引起性能问题是不可能的，因为我们正在执行计算的范围只是一个生产数据的简化子集。

- 为了简化，所有的值都要一次性推送到系统中。

  在用户验收测试中经常这样做，但是这会错过以不同速率投送数据时可能发生的关键预热和优化。

在用户验收测试中，大多数时候会通过简化测试数据集来让测试更容易。然而，这样的结果很少有用。

2. 评论示例

“让生产环境和用户验收测试环境保持同步太难了。”

“操控数据使其符合系统的期望太难了。”

“出于安全方面的考虑，生产数据是受保护的。开发人员不应该访问。”

3. 现实

为了得到精确的结果，用户验收测试环境中的数据必须与生产环境中的数据类似。如果出于安全原因数据无法使用，那么就应该对数据进行混淆，从而使其仍可以用于有意义的测试。另一个选择是对用户验收测试环境进行分区，虽然这样开发人员仍然看不到数据，但可以看到性能测试的输出，从而能够发现问题。

4. 讨论

这种反模式也落入了“有总比没有好”的陷阱。其想法是，即使是用过时的、不具代表性的数据来测试，也比不进行测试要好。

如前所述，这是一个极其危险的推理思路。在系统测试中，虽然对**某些东西**（即使它和

生产数据完全不一样）进行大规模测试可以发现缺陷和遗漏，但它提供的是一种虚假的安全感。

当系统上线后，使用模式并不符合通过用户验收测试数据锚定过的预期模式，这时开发和运维团队很可能会发现，他们还在为在用户验收测试环境中取得的胜利而沾沾自喜，对即将随大规模生产发布而来的恐怖后果毫无准备。

5. 解决方法

- 咨询数据领域的专家，并投入资源来将生产数据迁移到用户验收测试环境，必要时可以对数据进行混淆。
- 对预计有大量客户或交易的发布做好充足准备。

4.5 认知偏差与性能测试

即使面对一个可以借鉴过去经验和类似情况的问题，人类也可能无法迅速形成准确的意见。

> 认知偏差是一种会导致人脑得出错误结论的心理效应。问题的重点在于表现出这种偏差的人通常意识不到这一点，而且可能认为自己是理性的。

本章所探讨的许多反模式或多或少就是由一种或多种认知偏差造成的，而这些偏差又是基于某种无意识的假设产生的。

例如，对于“把责任归咎给驴”反模式，如果一个组件最近造成了几次故障，那么团队可能会偏向于认为任何新的性能问题都是由这个组件导致的。在分析过的数据中，如果有任何数据能够证实这个组件就是导致性能问题的罪魁祸首，那我们就会认为这样的数据更为可信。这种反模式结合了被称为确认偏差和近因偏差（一种假设最近已经发生的事情还会继续发生的倾向）的各个方面。

> Java 中的单个组件在不同的应用程序中会有不同的表现，这取决于它在运行时会被如何优化。为了消除任何预先存在的偏差，将应用程序作为一个整体来看待是很重要的。

有些偏差是互补的，有些是对称的。例如，有些开发人员可能会有这样的偏差，即认为问题根本与软件无关，它肯定出在软件所运行的基础设施上。这在“对我来说很有效”（works for me）反模式中很常见，其特征是认同诸如“这在用户验收测试环境中很好用，所以一定是生产套件出了问题”这样的观点。与之相反的认知偏差则认为每个问题一定是由软件引起的，因为那是开发人员了解并能直接影响系统的部分。

4.5.1 还原论思维

这种认知偏差基于一种分析方法，它坚持认为，如果能把一个系统分成足够小的若干部分，就可以通过理解它的组成部分来理解它。理解每个部分意味着减少了做出错误判断的概率。

以上观点的问题在于该分析方法无法应用于复杂的系统。有用的软件系统（或物理系统）几乎总是会出现突发行为，而整体又要大于各个部分的简单累加。

4.5.2 确认偏差

当涉及性能测试或试图主观地看待一个应用程序时，确认偏差会导致严重问题。当选择了一个糟糕的测试集或对测试结果没有进行统计分析时，就会无意识地引入确认偏差。确认偏差很难克服，因为通常有很强的动机或情绪因素在起作用（比如团队中有人试图证明某个观点）。

考虑一个反模式，比如“被热门技术分心”，有个团队成员希望引入最新、最好的 NoSQL 数据库。他们针对与生产数据并不相似的数据运行了一些测试，因为就评估目的而言，要表示完整的模式实在太复杂了。结果证明，在某个测试集上，NoSQL 数据库在他们本地机器上的访问时间非常出色。开发人员会将这一结果告诉每个人，并着手进行全面实施。这里有几个反模式在起作用，所有这些都会导致在新的库栈中出现新的、未经验证的假设。

摆弄开关

“按照坊间传说调优”和“忽略大局”（滥用微基准测试）都是反模式的例子，这些反模式至少在一定程度上是由还原论思维和确认偏差造成的。一个臭名昭著的例子是“按照坊间传说调优”的一个子类型，也就是所谓的“摆弄开关”。

这种反模式产生的原因是，虽然虚拟机尝试选择适合于所检测到的硬件的设置，但在某些情况下，工程师需要手动设置一些标志来调优代码的性能。这本身并无害处，但是由于 JVM 拥有很多命令行开关，具有高度可配置性，因此它也隐藏着一个认知陷阱。

可以使用如下开关来查看虚拟机标志的列表：

```
-XX:+PrintFlagsFinal
```

截至 Java 8u131，我们会得到 700 多个可能的开关。不仅如此，还有一些额外的调优选项，只有虚拟机在诊断模式下运行的时候才能使用。要查看这些选项，请添加这个开关：

```
-XX:+UnlockDiagnosticVMOptions
```

这样大约可以解锁另外 100 个开关。要想正确推断出应用这些开关的各种可能组合情况会产生的聚合效果，已经非人力所能及了。此外，在大多数情况下，实验观测表明改变开关值的影响很小——通常小于开发人员的预期。

4.5.3 战争的迷雾（行动偏差）

这种偏差通常会在服务中断或系统性能不如预期的情况下表现出来。最常见的原因包括：

- 更改了系统运行所基于的基础设施，可能没有收到通知或没有意识到会产生影响；
- 更改了系统所依赖的库；
- 在一年中业务最忙碌的一天出现了一个奇怪的 bug 或竞争条件。

在一个维护良好的应用程序中，如果有足够的日志记录和监控信息，那它们应该会生成清晰的错误信息，从而引导团队找出问题的根源。

然而，太多的应用程序没有测试过失败场景，也缺乏适当的日志记录。在这种情况下，即使是有经验的工程师也会掉入这样的陷阱，即认为他们自己在工作，以期解决服务中断问题，他们错把行动当成了速度——“战争的迷雾”降临了。

在这个时候，如果参与者对问题的处理方法没有系统化的思考，那么本章所讨论的许多人为因素就会粉墨登场了。例如，像“把责任归咎给驴”这样的反模式就会打断全面调查，把产品团队引向特定的调查路径——这往往会忽略大局。同样，团队可能会将系统分解成若干个组成部分，并在没有先确定问题真正存在于哪个子系统的情况下，就从低级的代码查起。

过去，使用系统化的方法来处理服务中断场景，把无须修补的东西留到事后调查时再处理，总是有回报的。然而，在人类情感方面，想把紧张的情绪从情境中抽离出来非常困难，尤其是面对服务中断的时候。

4.5.4 风险偏差

人类天生厌恶风险，抗拒改变，这主要是因为人们见过不少改变可能出错的例子，因此他们会尽量避免这种风险。但是当人们发现接受一些小的、经过计算的风险有可能推动产品的发展时，他们会感到非常沮丧。我们可以通过一组健壮的单元测试和产品回归测试来显著降低风险偏差。但是，如果其中任何一个测试都不值得信赖，那么改变就会变得非常困难，风险因素也无法控制。

这种偏差往往表现为未能从应用程序问题（甚至是服务中断）中吸取教训并采取适当的缓解措施。

4.5.5 埃尔斯伯格悖论

为了说明人类在理解概率方面有多糟糕，下面来看看埃尔斯伯格悖论。这个悖论的意思是，与“未知的未知”（unknown unknowns）相比，人类更喜欢“已知的未知”（known unknowns）。

埃尔斯伯格悖论的一般形式是一个简单的概率思维实验。在一个缸里有 90 个不同颜色的球，已知有 30 个蓝色球，其余的是红色球或绿色球。红色球和绿色球的确切分布是未知的，但在整个过程中，缸、球以及由此产生的胜率是固定的。

悖论的第一步是选择下注。玩家可以选择两种投注方式中的任何一种。

A. 如果随机抽到的是蓝色球，玩家将赢得 100 美元。

B. 如果随机抽到的是红色球，玩家将赢得 100 美元。

大多数人选择 A，因为它代表的是已知的胜率：中奖的可能性正好是 1/3。假设一个球被抽出后会被放回同一个缸里并重新打乱顺序，这时候再让玩家来选一次。令人惊讶的事情发生了，这次的两个选项如下。

C. 如果随机抽到的是蓝色球或绿色球，玩家将赢得 100 美元。

D. 如果随机抽到的是红色球或绿色球，玩家将赢得 100 美元。

在这种情况下，投注 D 对应的是已知的胜率（2/3 的中奖概率），所以基本上每个人都会选择 D。

悖论是，选择 A 和 D 的组合并不理性。选择 A，隐含着对红色球和绿色球分布情况的看法，即认为“绿色球比红色球多”。因此，如果选择了 A，那么合理的策略是选择 C 与其搭配，因为与安全地选择 D 相比，选择 C 的胜率更高。

4.6　小结

当评估性能结果时，一定要以恰当的方式处理数据，避免陷入不科学、主观的思考中。本章介绍了一些测试类型、测试最佳实践以及性能分析中伴生的反模式。

下一章将研究底层的性能测量手段、微基准测试的陷阱，以及一些用于处理从 JVM 中测得的原始结果的统计技术。

第 5 章

微基准测试与统计

本章将考虑直接测量 Java 性能数字的具体细节。JVM 的动态特性意味着性能数字往往比许多开发人员预期的要更难处理。因此，互联网上出现了许多不准确或带有误导性的性能数字。

本章的一个主要目标是确保你意识到这些可能的陷阱，并且只生成你和其他人可以信赖的性能数字。特别需要注意的是，对小块 Java 代码的测量（**微基准测试**）非常微妙且难以正确完成，这也是本章将要探究的主要内容，同时我们还会介绍性能工程师应该如何正确使用它。

> 你不能糊弄自己，你自己恰恰是最容易被蒙骗的人。
>
> ——Richard Feynman

接下来，本章会介绍如何使用微基准测试工具的黄金标准：JMH。在考虑过所有的警告和说明之后，如果你真的觉得你的应用程序和用例需要使用微基准测试，那么从现有的最可靠、最先进的工具入手，可以让你避免很多众所周知的陷阱。

最后，我们会再来谈谈统计。JVM 通常会产生一些需要谨慎对待的性能数字。因为微基准测试生成的数字通常特别灵敏，所以需要性能工程师用一定程度的统计思维来处理观察到的结果。本章最后几节将介绍一些处理 JVM 性能数据的技术以及关于如何解释数据的问题。

5.1 Java性能测量

我们在 1.3 节曾描述过性能分析是几个不同方面的组合体，从而使其从根本上说是一门实验科学。

也就是说，如果我们想写出一个好的基准测试（或微基准测试），那么把它看成一个科学

实验是非常有帮助的。

这种方法让我们将基准测试看成一个有输入和输出的“黑盒子”，希望从中收集能帮助推测结果的数据。然而，我们必须谨慎行事，因为仅仅**收集**数据是不够的，我们还需要确保不会被数据**欺骗**。

> 基准测试数字本身并不重要，重要的是你从这些数据中推导出了什么样的模型。
>
> ——Aleksey Shipilëv

因此，我们的理想目标是保证基准测试的公平性，也就是说，要尽可能只改变系统的一个方面，并确保基准测试中的其他任何外部因素都是可控的。在理想的世界中，系统的其他可能改变的方面在多个测试之间是完全不变的，但在实践中我们很少如此幸运。

> 即使在实践中无法实现科学意义上纯粹公平的测试这一目标，我们的基准测试也至少要保证可重复性，因为这是任何实验结果的基础。

编写 Java 平台基准测试的一个核心问题是 Java 运行时的复杂性。本书有相当大的篇幅就是用来阐释 JVM 对开发人员的代码所做的自动优化。当把基准测试看作这些优化背景下的科学测试时，我们的选择就很有限了。

也就是说，要完全理解并说明这些优化的精确影响几乎是不可能的。应用程序代码的“真实”性能的精确模型很难建立，而且其适用性也往往有限。

换句话说，我们无法将执行中的 Java 代码与 JIT 编译器、内存管理以及 Java 运行时提供的其他子系统真正分开。我们也不能忽略运行测试时当前的操作系统、硬件或运行时条件（比如负载）的影响。

> 没有谁是一座孤岛。
>
> ——John Donne

处理规模较大（整个系统或子系统）的问题更容易消除这些影响。相反，当处理规模较小的问题或进行微基准测试时，真正将应用程序代码与运行时的后台行为隔离开来会更加困难。正如本章将要讨论的，这就是微基准测试如此困难的根本原因。

下面来考虑一个看似非常简单的例子。有段代码要对 100 000 个数字组成的列表进行排序，我们对它做基准测试，希望从尝试创建一个真正公平的测试的角度进行检验。

```
public class ClassicSort {

    private static final int N = 1_000;
    private static final int I = 150_000;
    private static final List<Integer> testData = new ArrayList<>();

    public static void main(String[] args) {
        Random randomGenerator = new Random();
        for (int i = 0; i < N; i++) {
            testData.add(randomGenerator.nextInt(Integer.MAX_VALUE));
```

```
        }

        System.out.println("Testing Sort Algorithm");

        double startTime = System.nanoTime();

        for (int i = 0; i < I; i++) {
            List<Integer> copy = new ArrayList<Integer>(testData);
            Collections.sort(copy);
        }

        double endTime = System.nanoTime();
        double timePerOperation = ((endTime - startTime) / (1_000_000_000L * I));
        System.out.println("Result: " + (1 / timePerOperation) + " op/s");
    }
}
```

该基准测试创建了一个由随机整型数组成的数组，一旦创建完成，就会记录基准测试的开始时间。然后，基准测试会执行循环，复制模板数组，再对数据进行排序。一旦运行完 I 次，持续时间就会被转换为秒，然后除以迭代次数，从而得到每次操作所花的时间。

这个基准测试的第一个问题是，它没有考虑预热 JVM 就直接测试代码了。考虑一下排序在生产中的服务器应用程序中运行的情况。它很可能已经运行了数小时，甚至数天。然而我们知道 JVM 包含一个 JIT 编译器，它会将解释的字节码转换为高度优化的机器代码。这个编译器在方法运行了一定次数后才会生效。

因此，我们正在进行的测试并不能代表它在生产中的表现。在我们尝试做基准测试的时候，JVM 会花时间优化这个调用。使用几个 JVM 标志来运行该排序，可以看到以下这个效果：

```
java -Xms2048m -Xmx2048m -XX:+PrintCompilation ClassicSort
```

`-Xms` 和 `-Xmx` 选项用于控制堆的大小，本例中堆的大小被固定为 2 GB。`PrintCompilation` 的作用是，每当一个方法被编译（或者发生了其他编译事件），JVM 就会输出一行日志。下面是输出的一个片段：

```
Testing Sort Algorithm
  73    29   3   java.util.ArrayList::ensureExplicitCapacity (26 bytes)
  73    31   3   java.lang.Integer::valueOf (32 bytes)
  74    32   3   java.util.concurrent.atomic.AtomicLong::get (5 bytes)
  74    33   3   java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
  74    35   3   java.util.Random::next (47 bytes)
  74    36   3   java.lang.Integer::compareTo (9 bytes)
  74    38   3   java.lang.Integer::compare (20 bytes)
  74    37   3   java.lang.Integer::compareTo (12 bytes)
  74    39   4   java.lang.Integer::compareTo (9 bytes)
  75    36   3   java.lang.Integer::compareTo (9 bytes) made not entrant
  76    40   3   java.util.ComparableTimSort::binarySort (223 bytes)
  77    41   3   java.util.ComparableTimSort::mergeLo (656 bytes)
  79    42   3   java.util.ComparableTimSort::countRunAndMakeAscending (123 bytes)
  79    45   3   java.util.ComparableTimSort::gallopRight (327 bytes)
  80    43   3   java.util.ComparableTimSort::pushRun (31 bytes)
```

JIT 编译器会全力以赴地优化调用层次的不同部分，以提高代码的效率。这意味着基准测试的性能会随着抓取计时信息持续的时间而变化，而且由于我们不经意间在实验中留下了一个不受控制的变量，因此需要一个预热阶段，让 JVM 在我们抓取计时信息之前稳定下来。通常的做法是，先将要进行基准测试的代码运行若干次，但不抓取计时信息。

我们需要考虑的另一个外部因素是垃圾收集。理想情况下，我们希望垃圾收集在计时过程中不要运行，而且希望通过设置使其行为标准化。但是由于垃圾收集的不确定性，这是难以控制的。

我们可以做的一个改进是确保在垃圾收集可能运行的时候不去获取计时信息。我们可以要求系统运行一次垃圾收集并稍等片刻，但系统可能会决定忽略这个调用。就目前而言，这个基准测试中的计时太过宽泛了，所以我们需要更多关于可能发生的垃圾收集事件的细节。

不仅如此，除了选择计时点之外，我们还希望选择一个合理的迭代次数，而要通过试验和改进找出这个数可能有点困难。垃圾收集的作用可以通过另一个虚拟机标志来查看（关于日志格式的详细内容，参见第 7 章）：

```
java -Xms2048m -Xmx2048m -verbose:gc ClassicSort
```

它将产生类似下面这样的垃圾收集日志条目：

```
Testing Sort Algorithm
[GC (Allocation Failure)  524800K->632K(2010112K), 0.0009038 secs]
[GC (Allocation Failure)  525432K->672K(2010112K), 0.0008671 secs]
Result: 9838.556465303362 op/s
```

基准测试中另一个常见错误是，我们正在测试的代码所生成的结果实际上并没有被用到。在上面的基准测试中，`copy` 实际上是死代码，所以有可能被 JIT 编译器识别为死代码路径并优化掉，而事实上这正是要进行基准测试的地方。

进一步要考虑的是，即使是平均值，查看单个计时结果也无法全面了解基准测试的性能。理想情况下，我们希望捕捉到误差幅度（margin of error），以了解所收集值的可靠性。如果误差幅度很高，则它可能指向某个不可控的变量，或者确实是我们写的代码性能不高。不管是哪种情况，如果不捕捉到误差幅度，就无法确定是否存在问题。

即使是对一个非常简单的排序进行基准测试也可能有一些陷阱，这就意味着这个基准测试就要被否决。然而，随着复杂性的增加，事情就会迅速变得越来越糟。考虑一个评估多线程代码的基准测试。多线程代码极难进行基准测试，因为它需要确保从基准测试开始到获得特定的精确结果的过程中，所有线程都要挂着等待直到彼此完全启动。如果做不到这一点，误差幅度就会很高。

在对并发代码进行基准测试时，还要考虑硬件情况，而且不仅仅是硬件配置，比如要考虑电源管理是否会生效，或者机器上是否有其他资源争用情况。

想要把基准测试代码写正确是非常复杂的，而且需要考虑很多因素。作为开发人员，我们最关心的应该是所要剖析的代码，而不是刚才强调的所有问题。上述所有问题共同导致了这样一种情况：除非你是 JVM 专家，否则很容易漏掉一些东西并得到一个错误的基准测试结果。

有两种方法可以解决这个问题。其一是只对整个系统进行基准测试。在这种情况下，底层的数字通常会被忽略掉，而不会被收集起来。对多次独立运行的结果取平均值可以获得有意义的大规模结果，大多数开发人员需要的往往就是这种方法。

其二是试图通过一个通用的框架来解决上述的许多问题，以便对相关底层结果进行有意义的对比。理想的框架可以消除刚刚讨论到的一些压力。这样的工具必须要紧跟 OpenJDK 的主线开发，以确保新的优化和其他外部控制变量都能得到管理。

幸运的是，这样的工具确实存在，它也是下一节的主题。然而，对于大多数开发人员来说，可以纯粹将其当作参考资料，直接跳到 5.3 节继续学习。

5.2 JMH

接下来我们会用一个具有警示性的例子来说明为什么简单设计的微基准测试很容易出错以及为什么出错。从中我们也获得了一些启发，有助于我们判断自己的用例是否适合微基准测试，同时得到的结论是，该技术并不适合绝大多数的应用程序。

5.2.1 不是万不得已，不要做微基准测试（一个真实的故事）

在办公室待了漫长的一天之后，本书的一位作者在正要离开大楼时看到一位女同事仍在伏案工作，紧盯着一个 Java 方法。当时他也没多想，就去赶火车回家了。然而两天之后，类似的场景又出现了，那位同事的计算机屏幕上还是一个非常相似的方法，她的脸上满是疲惫和厌烦。很明显需要进行更深入地调查了。

她正在维护的应用程序有一个很明显的性能问题。尽管使用了较新版本的知名库，但新版本的应用程序性能还不如团队想要替换掉的旧版本。她花了一些时间删除部分代码并写了一些小的基准测试，试图找到问题所在。

说不出为什么，就感觉这种方法不对，就像大海捞针一样。他们又一起尝试了另一种方法，并很快确认该应用程序的 CPU 利用率已经达到了最大值。因为知道这种用例适合用执行剖析器（关于何时使用剖析器的完整细节，参见第 13 章），所以我们用了 10 分钟对应用程序进行剖析，然后就找到了真正的原因。果然，问题根本不在应用程序代码中，而是在团队正在使用的一个新的基础设施库中。

这个故事说明了一种处理 Java 性能问题的方法，不幸的是，这种情况太常见了。开发人员可能会一度认为一定是自己代码的问题，而错过了大局。

> 开发人员往往希望通过仔细观察小规模的代码构造来开始寻找问题，但在这个层面上进行基准测试极为困难，而且有一些危险的“陷阱”。

5.2.2 关于何时使用微基准测试的启发

正如在第 2 章中曾简单讨论的，Java 平台的动态特性以及垃圾收集和激进的 JIT 优化这样

的特性，导致我们很难直接找到出现性能问题的原因。更糟糕的是，性能数字往往取决于应用程序被测量时的确切运行时情况。

> 分析整个 Java 应用程序的真实性能总是比分析一小段 Java 代码更容易。

然而，有时我们需要直接分析单个方法甚至单个代码片段的性能。不过这种分析也不能掉以轻心。一般来说，底层分析或微基准测试主要有以下 3 种应用场景。

- 你正在开发应用广泛的通用库代码。
- 你是 OpenJDK 或其他 Java 平台实现的开发人员。
- 你正在开发对延迟极为敏感的代码（比如用于低延迟交易）。

这 3 种应用场景背后的原理略有不同。

通用型库（顾名思义）对其使用环境所知甚少。这类库的示例包括 Google Guava 或 Eclipse Collections（最初由高盛贡献）。它们需要在非常广泛的用例（从包含几十个元素的数据集到亿级数据集）中提供可接受的或更好的性能。

由于应用非常广泛，通用库有时不得不使用微基准测试来进行较传统的性能和容量测试。

平台开发人员是微基准测试的主要用户群体，OpenJDK 团队创建的 JMH 工具主要就是供他们自己使用的。不过事实证明，这个工具对更广泛的性能专家群体也是很有用的。

最后，有一些从事 Java 性能前沿工作的开发人员可能希望使用微基准测试来选择最适合其应用程序和极端用例的算法和技术。这将包括低延迟的金融交易以及少量的其他案例。

如果你是 OpenJDK 或通用库的开发人员，那么是否进行微基准测试就已经很明显了；但可能有些开发人员还是会迷惑，他们的性能需求是不是应该考虑微基准测试。

> 微基准测试的可怕之处在于，它们总是会给出一个数字，即使这个数字毫无意义。它们确实测量了某个事物，只是我们并不确定测量的是什么。
>
> ——Brian Goetz

一般来说，只有最极端的应用程序才应该使用微基准测试。虽然没有明确的规则，但除非你的应用程序符合以下大部分或全部标准，否则你不太可能从应用程序的微基准测试中获得真正的好处。

- 你的总代码路径执行时间一定要小于 1 毫秒，并且很可能要小于 100 微秒。
- 你应该已经测量过你的内存（对象）分配率（详见第 6 章和第 7 章），它应该小于 1 MB/s，最好是非常接近于零。
- 你应该使用了接近 100% 的可用 CPU，而且系统利用率应该一直比较低（低于 10%）。
- 你应该已经使用了执行剖析器（参见第 13 章）来了解正在消耗 CPU 的方法的分布情况。分布中最多应该有 2~3 种主要方法。

综上所述，微基准测试显然是一种高级且很少使用的技术。然而，了解它所反映出的一些

基本理论和复杂性是很有帮助的，因为这能让我们更好地了解处理 Java 平台上不太极端的应用程序的性能也是有很多困难的。

> 任何没有进行反汇编和代码生成分析的 Nano 级别基准测试都是不可信的。言尽于此。
>
> ——Aleksey Shipilëv

本节剩余篇幅会更深入地探讨微基准测试，并介绍一些工具以及开发人员必须考虑的事项，以便生成可靠的结果，而不会导致错误的结论。不管它是否与你当前的项目直接相关，这些信息对性能分析人员来说都是很有用的背景知识。

5.2.3 JMH框架

JMH 是为解决我们刚才讨论的问题而设计的框架。

> JMH 是一个 Java 工具，用于构建、运行和分析用 Java 和其他以 JVM 为目标平台的语言编写 nano/micro/milli/macro 基准测试。
>
> ——OpenJDK

过去已经有一些对简单的基准测试库的尝试，其中 Google Caliper 是最受开发人员欢迎的一个。然而，所有这些框架都有其挑战，而且往往看似合理的设置或测量代码性能的方式通常会有一些微妙的陷阱需要应对。特别是随着 JVM 的不断演进，应用新的优化方法时，情况更是如此。

JMH 在这方面有很大的不同，它就是由构建 JVM 的同一群工程师开发的。因此，JMH 的作者们知道如何避免 JVM 的每个版本中存在的优化陷阱。它作为一个基准测试工具，会随着每个 JVM 版本的发布而演进，所以开发人员只需要关注这个工具的使用以及基准测试代码本身。

除了一些已经强调的问题外，JMH 还考虑到了一些关键的基准测试安全设计问题。

基准测试框架必须是动态的，因为它在编译时不知道基准测试的内容。要解决这个问题，明显可以选择使用反射来执行用户编写的基准测试。然而，这样又会在基准测试的执行路径中引入另一个复杂的 JVM 子系统。相反，JMH 通过注解处理从基准测试中生成额外的 Java 源代码来进行操作。

> 由于许多常见的、基于注解的 Java 框架（如 JUnit）使用反射来实现其目标，因此使用一个处理程序来生成额外的源代码可能会让一些 Java 开发人员有些意外。

一个问题是，如果基准测试框架调用用户的代码以进行大量迭代，那么可能就会触发循环优化。这意味着运行基准测试的实际过程可能会影响结果的可靠性。

为了避免触发循环优化，JMH 会为基准测试生成代码，将基准测试代码封装在一个循环中，并小心地将迭代次数设置为一个能避免优化的值。

5.2.4　执行基准测试

JMH 执行所涉及的复杂性对用户而言大部分是隐藏的，而且使用 Maven 来设置一个简单的基准测试相当直接。可以通过执行以下命令来设置一个新的 JMH 项目：

```
$ mvn archetype:generate \
          -DinteractiveMode=false \
          -DarchetypeGroupId=org.openjdk.jmh \
          -DarchetypeArtifactId=jmh-java-benchmark-archetype \
          -DgroupId=org.sample \
          -DartifactId=test \
          -Dversion=1.0
```

这会下载所需的资源，并创建一个用来放置代码的存根（stub）。

基准测试使用了 `@Benchmark` 来注解，表示 JMH 将执行该方法来进行基准测试（在框架执行了各种设置任务后）：

```
public class MyBenchmark {
    @Benchmark
    public void testMethod() {
        // 代码存根
    }
}
```

基准测试的作者可以通过配置参数来控制其执行，这些参数可以在命令行或基准测试的 `main()` 方法中设置，如下所示：

```
public class MyBenchmark {

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(SortBenchmark.class.getSimpleName())
                .warmupIterations(100)
                .measurementIterations(5).forks(1)
                .jvmArgs("-server", "-Xms2048m", "-Xmx2048m").build();

        new Runner(opt).run();
    }
}
```

命令行上的参数会覆盖 `main()` 方法中设置的任何参数。

通常情况下，基准测试都需要一些设置，比如创建一个数据集，或者为一组用来比较性能的、正交的基准测试集设置所需的条件。

状态，以及控制状态是 JMH 框架所具有的另一个特性。可以用 `@State` 注解来定义该状态，并接受 `Scope` 枚举来定义状态的可见范围：`Benchmark`、`Group` 还是 `Thread`。使用 `@State` 注解的对象在基准测试的生命周期内都是可达的，它可能还需要执行一些必要的设置。

多线程代码也需要谨慎处理，以确保基准测试不会因为状态管理不善而出现偏差。

一般来说，如果在一个方法中执行的代码没有副作用并且返回结果也没有用到，那 JVM 就可以考虑移除这个方法。JMH 需要防止这种情况的发生，而且事实上，对于基准测试的

作者来说，这种情况带来的影响是非常直接的。如果是单个结果，那么基准测试方法可以将其返回，而且该框架要确保暗中将这个值赋值到某个地方（黑洞），这是框架作者开发的一种机制，其性能开销可以忽略不计。

如果一个基准测试要执行多个计算，那么将该方法的结果组合起来并返回的成本可能会很高。在这种情况下，作者可能需要使用一个显式的黑洞，具体做法是创建一个以 `Blackhole` 为参数的基准测试。

`Blackhole` 提供了 4 种与优化相关的保护，以避免这些优化影响基准测试。一些保护是阻止基准测试因为其作用范围有限而产生的过度优化，其他一些是避免数据存在可预测的运行时模式，而这些模式在系统的典型运行情况下是不会出现的。这些保护是：

- 防止运行时把死代码优化掉；
- 防止重复计算被折叠成常量；
- 防止伪共享，读写某个值会影响当前的高速缓存行；
- 避免“写墙”（write wall）。

性能中的**墙**（wall）这个术语通常指的是资源已经饱和并且事实上成了应用程序的瓶颈。如果命中了“写墙”，就会影响高速缓存，并污染用于写入的缓冲区。如果在自己的基准测试内出现了这种情况，那么可能会带来极大的影响。

正如 `Blackhole` 的 JavaDoc 中所写的那样（如前所述），为了提供这些保护，你必须对 JIT 编译器有深入的了解才可能构建出避免优化的基准测试。

让我们来看看 `Blackhole` 使用的两个 `consume()` 方法，以了解 JMH 使用的一些技巧（如果你对 JMH 的实现方式不感兴趣，可以跳过）。下面是第一个方法：

```
public volatile int i1 = 1, i2 = 2;

/**
 * 消耗对象。该调用有一个副作用，可以防止JIT消除我们要依赖的计算
 *
 * @param i要消耗的int
 */
public final void consume(int i) {
    if (i == i1 & i == i2) {
      // 不应该发生
        nullBait.i1 = i; // 隐含空指针异常
    }
}
```

重复编写这段代码来支持所有基本类型（就是把 `int` 改为相应的基本类型）。变量 `i1` 和 `i2` 被声明为 `volatile`，这意味着运行时必须重新计算。虽然 `if` 语句永远不可能为真，但编译器必须允许代码运行。我们还要注意在 `if` 语句中使用了按位与运算符（`&`）。这不仅避免了额外的分支逻辑带来的问题，还带来了更一致的性能。

下面是第二个方法：

```
    public int tlr = (int) System.nanoTime();

    /**
     * 消耗对象。该调用有一个副作用，可以防止JIT消除我们要依赖的计算
     *
     * @param obj要消耗的对象
     */
    public final void consume(Object obj) {
            int tlr = (this.tlr = (this.tlr * 1664525 + 1013904223));
        if ((tlr & tlrMask) == 0) {
         // 在测量中几乎不应该发生
            this.obj1 = obj;
            this.tlrMask = (this.tlrMask << 1) + 1;
        }
    }
```

当处理对象时，起初似乎可以使用相同的逻辑，因为用户的对象也不会与 Blackhole 持有的对象相同。然而，编译器也想表现得聪明一点。如果编译器通过逃逸分析断言这个对象不会和其他对象等价，那么比较操作本身就有可能直接被优化为返回 false。

相反，对象只在极少数情况才会执行的条件下会被消耗。计算出 tlr 的值，并将其与 tlrMask 进行按位与运算，以减少 0 值出现的概率，但不会彻底消除这种概率。这样可以保证对象能在很大程度上被消耗掉，而不需要重新赋值。基准测试框架的代码读起来是非常有趣的，因为它与现实世界中的 Java 应用程序有很大的区别。事实上，如果在生产 Java 应用程序中的任何地方发现了这样的代码，则很可能会解雇负责的开发人员。

除了编写一个极其精确的微基准测试工具之外，JMH 作者们还为其中的类创建了令人印象深刻的文档。如果你对幕后的神奇之处感兴趣，那么注释可以很好地说明这一点。

通过前面的信息，我们知道不需要太多的工作就能启动和运行一个简单的基准测试，但 JMH 也有一些相当高级的功能。官方文档为每个特性提供了示例，所有这些都值得一看。

展示 JMH 的强大功能及其与 JVM 比较接近的有趣特性包括以下内容：

- 能够控制编译器；
- 在基准测试期间模拟 CPU 利用水平。

另一个很好的特性是使用 Blackhole 来实际消耗 CPU 周期，从而使你可以模拟不同 CPU 负载下的基准测试。

@CompilerControl 注解可用于要求编译器不要内联、明确内联或者从编译中排除该方法。如果遇到性能问题，而你又怀疑是由 JVM 的内联或编译引起的，则下面这个注解非常有用：

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(1)
public class SortBenchmark {
```

```java
    private static final int N = 1_000;
    private static final List<Integer> testData = new ArrayList<>();

    @Setup
    public static final void setup() {
        Random randomGenerator = new Random();
        for (int i = 0; i < N; i++) {
            testData.add(randomGenerator.nextInt(Integer.MAX_VALUE));
        }
        System.out.println("Setup Complete");
    }

    @Benchmark
    public List<Integer> classicSort() {
        List<Integer> copy = new ArrayList<Integer>(testData);
        Collections.sort(copy);
        return copy;
    }

    @Benchmark
    public List<Integer> standardSort() {
        return testData.stream().sorted().collect(Collectors.toList());
    }

    @Benchmark
    public List<Integer> parallelSort() {
        return testData.parallelStream().sorted().collect(Collectors.toList());
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(SortBenchmark.class.getSimpleName())
                .warmupIterations(100)
                .measurementIterations(5).forks(1)
                .jvmArgs("-server", "-Xms2048m", "-Xmx2048m")
                .addProfiler(GCProfiler.class)
                .addProfiler(StackProfiler.class)
                .build();

        new Runner(opt).run();
    }
}
```

运行该基准测试会产生如下输出：

```
Benchmark                          Mode  Cnt      Score      Error  Units
optjava.SortBenchmark.classicSort  thrpt  200  14373.039 ±  111.586  ops/s
optjava.SortBenchmark.parallelSort thrpt  200   7917.702 ±   87.757  ops/s
optjava.SortBenchmark.standardSort thrpt  200  12656.107 ±   84.849  ops/s
```

查看这个基准测试，你可能很快就得出结论，即经典的排序方法比使用流更高效。两个代码运行都使用了一个数组副本和一次排序，所以该基准测试应该没问题。开发人员可能看到低错误率和高吞吐量就认为这个基准测试一定正确。

但是让我们考虑一下基准测试可能未能准确反映性能的一些原因，也就是要弄清楚它是否是一个受控的测试。首先我们看一下垃圾收集对 `classicSort` 测试的影响：

```
Iteration   1:
[GC (Allocation Failure)  65496K->1480K(239104K), 0.0012473 secs]
[GC (Allocation Failure)  63944K->1496K(237056K), 0.0013170 secs]
10830.105 ops/s
Iteration   2:
[GC (Allocation Failure)  62936K->1680K(236032K), 0.0004776 secs]
10951.704 ops/s
```

在这个快照中，很明显大约每次迭代都会发生一个垃圾收集周期，将其与下面这个并行排序相比较会很有趣：

```
Iteration   1:
[GC (Allocation Failure)  52952K->1848K(225792K), 0.0005354 secs]
[GC (Allocation Failure)  52024K->1848K(226816K), 0.0005341 secs]
[GC (Allocation Failure)  51000K->1784K(223744K), 0.0005509 secs]
[GC (Allocation Failure)  49912K->1784K(225280K), 0.0003952 secs]
9526.212 ops/s
Iteration   2:
[GC (Allocation Failure)  49400K->1912K(222720K), 0.0005589 secs]
[GC (Allocation Failure)  49016K->1832K(223744K), 0.0004594 secs]
[GC (Allocation Failure)  48424K->1864K(221696K), 0.0005370 secs]
[GC (Allocation Failure)  47944K->1832K(222720K), 0.0004966 secs]
[GC (Allocation Failure)  47400K->1864K(220672K), 0.0005004 secs]
```

所以，通过加入 JVM 标志来查看导致这种意外差异的原因，我们可以发现基准测试中的其他地方正在引起噪声（在本例中为垃圾收集）。

得出的结论就是，我们很容易假设基准测试代表一个受控的环境，但事实可能更加复杂。因为不可控的变量一般很难发现，所以即使有了 JMH 这样的工具，也要小心谨慎。我们还需要注意纠正自己的确认偏差，并确保所测量的观测变量能够真实反映系统的行为。

我们在第 9 章中将介绍 JITWatch，它将提供另一个关于 JIT 编译器如何处理字节码的视图，这通常有助于我们深入了解为什么为特定方法生成的字节码可能会导致基准测试无法按预期执行。

5.3　JVM性能统计

如果性能分析真的是一门实验科学，那么我们自己必然要处理结果数据的分布。统计学家和科学家都知道，来自现实世界的结果几乎从来不能用清晰明显的信号来表示。我们必须根据自己的所见真实地看待世界，而不是用我们想要找到的、过于理想化的世界来面对它。

> 我们信奉上帝。其他人请用数据说话。
>
> ——Michael Bloomberg

所有的测量都会包含一定量的误差。下一节将介绍 Java 开发人员在进行性能分析时可能会遇到的两种主要的误差类型。

5.3.1 误差类型

工程师可能会遇到的误差主要有两个来源。

系统性误差

某个不明因素正在以一种相关的方式影响可观测变量的测量。

随机性误差

某个测量误差或不相关的因素以不相关的方式影响了结果。

每种类型的误差都有特定的词汇。比如，**准确度**（accuracy）用来描述测量中系统性误差的程度，准确度越高，系统性误差越低。同样，随机性误差对应的术语是**精确度**（precision），精确度越高，随机性误差越低。

图 5-1 显示了某次测量中两类误差的效果。第一幅图显示的是一群测量结果围绕在真实结果（目标中心）周围，这些测量值具有很高的准确度和精确度。第二幅图显示的是有一个系统性影响（或许没有做好校准？）导致所有的测量都偏离目标，所以这些测量值精确度很高，但准确度很低。第三幅图显示的是测量基本对准了目标，但松散地聚集在中心附近，所以精确度低，但准确度高。最后一幅图显示的是测量没有给出明确的信号，所以精确度和准确度都不高。

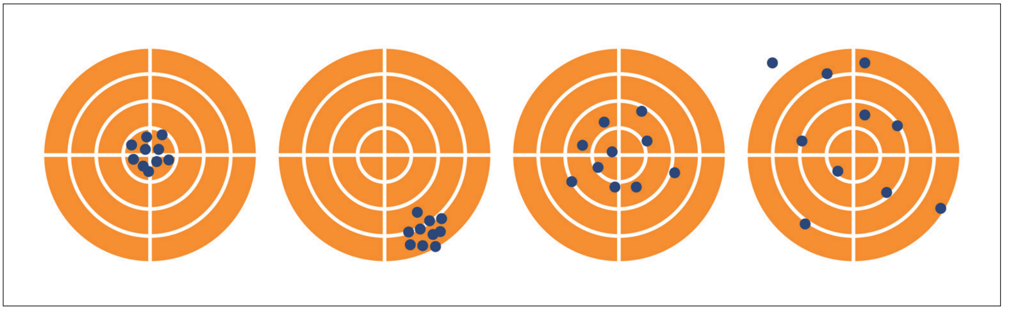


图 5-1：不同类型的误差

1. 系统性误差

例如，考虑针对一组发送和接收 JSON 文件的后端 Java Web 服务运行的性能测试。当直接使用应用程序前端进行负载测试而出现问题时，这类性能测试就很常见。

图 5-2 是从 Apache JMeter 负载生成工具的 JP GC 扩展包生成的。图中实际上有两个系统性的影响在起作用。第一个需要注意的影响是在最上面的那条线（异常的服务）中观察到的线性模式，它代表着某些有限的服务器资源在缓慢耗尽。这类模式通常与内存泄漏有关，或者是由于线程在请求处理期间使用了其他资源而没有释放，具体原因需要去调查研究，虽然看上去像是真正的问题所在。

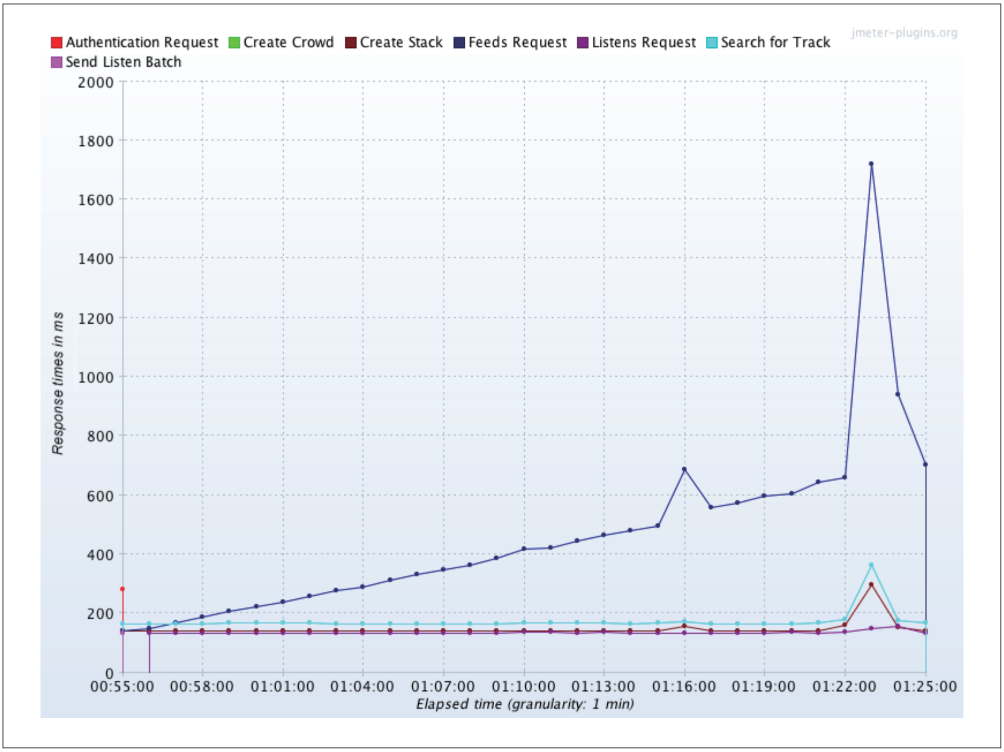


图 5-2：系统性误差

> 还需要进一步的分析来确认受影响的资源类型；我们不能简单地断定是内存泄漏。

第二个需要注意的影响是其他大多数服务在 180 毫秒左右表现出的一致性。这是很可疑的，因为这些服务在响应请求时所做的工作量大有不同，可是结果为什么会如此一致呢？

答案就是：虽然被测试的服务在英国伦敦部署，但负载测试是在印度的孟买进行的。观察到的响应时间包括了从孟买到伦敦必要的来回网络延迟。网络延迟在 120~150 毫秒的范围内，因此除了异常点之外，它占据了我们观察到的绝大部分服务时间。

这个巨大的系统性影响掩盖了实际响应时间的差异，因为这些服务的实际响应时间远小于 120 毫秒。这个系统性误差的例子并不代表应用程序存在问题，相反，是测试设置中出现了一个问题。好消息是，当在伦敦重新运行这个测试时，不出所料，这个误差就完全消失了。

2. 随机性误差

这里值得一提的是随机性误差的情况，尽管人们经常遇到。

这里的讨论是假定读者熟悉对符合正态分布的测量结果所采用的基本统计处理方法（均值、众数、标准差等）；不熟悉的读者可以查阅基础教科书，比如 *Handbook of Biological Statistics*。[1]

随机性误差是由环境中未知或不可预测的变化引起的。在一般的科学应用中，这些变化可能发生在测量仪器或环境中，但对于软件来说，因为我们假设测量保护是可靠的，所以随机性误差的来源只能是操作环境。

人们通常认为随机性误差遵循高斯分布（又叫正态分布）。图 5-3 演示了两个典型的例子。如果无论正负，误差相对于观测对象都具有同等的可能性，那么高斯分布就是一个很好的模型，但正如我们将看到的，该分布并不适合 JVM。

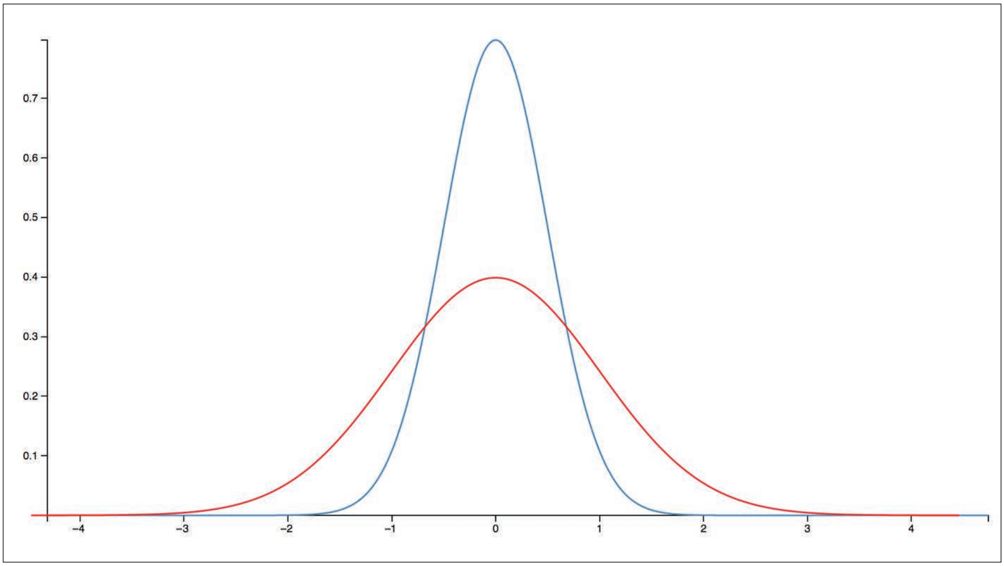


图 5-3：高斯分布（又称正态分布或钟形曲线）

3. 假相关

关于统计学，最著名的一个谚语是“相关性不代表因果性”，也就是说，两个变量的行为相似并不意味着它们之间有潜在的联系。

在最极端的例子中，如果工作人员足够努力地寻找，那么在完全不相关的测量结果之间也可以找到某种关联。比如图 5-4 显示出美国的鸡肉消费与原油进口总量有很好的相关性。[2]

注 1：*Handbook of Biological Statistics, 3rd ed*，作者是 John H. McDonald，由位于美国马里兰州巴尔的摩的 Sparky House Publishing 于 2014 年出版。

注 2：本节所举的假相关的例子来自 Tyler Vigen 网站，此处已经获得基于创作共用许可证（Creative Commons License）的授权允许重复使用。如果你喜欢，Vigen 网站上有一本书提供了更多有趣的例子，可以通过登录该网站查找。

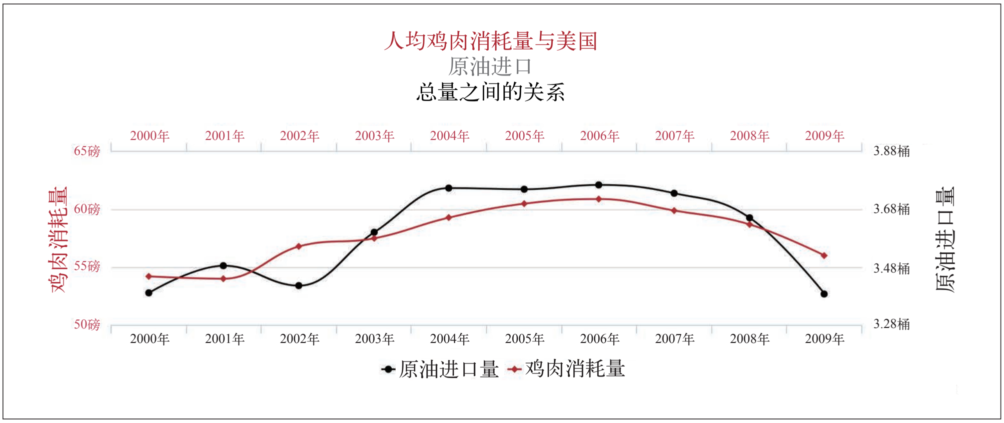


图 5-4：完全假相关的例子（Vigen）

这些数字之间显然没有因果关系。没有任何因素可以同时影响原油进口和鸡肉的食用。然而，从业者需要警惕的并不是这些荒谬可笑的相关性。

图 5-5 中显示的是游戏厅的收入与计算机科学博士学位授予数量的相关性。从中我们不难理解一项社会学研究声称观察到的这些数据之间存在着联系，或许可以认为“压力大的博士生会玩几个小时电子游戏寻求放松”。让人郁闷的是，尽管实际上并不存在这种相关因素，但这类声明仍然非常常见。

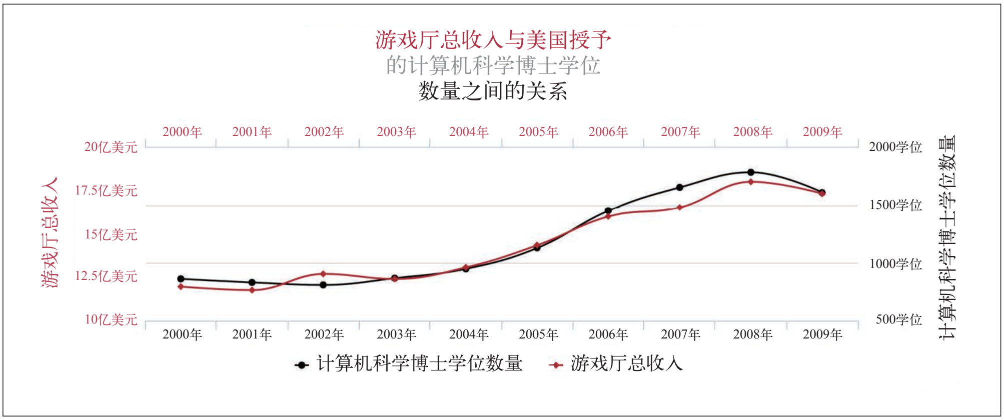


图 5-5：不那么假的相关性（Vigen）

在 JVM 和性能分析领域，需要注意的是，不要仅仅根据相关性以及“看似合理”的联系就在测量结果之间归结出因果关系。这也是 Feynman 的格言“你不能糊弄自己”所表达内容的一个方面。

我们已经见过一些误差来源的示例，也提到了臭名昭著的陷阱——假相关，那么接下来会讨论 JVM 性能测量中一个需要特别注意的方面。

5.3.2　非正态统计

基于正态分布的统计学不需要太多复杂的数学知识。因此，通常在大学预科或本科阶段所教授的标准统计学方法主要着重于正态分布数据的分析。

在该阶段，会教授学生计算均值和标准差（或方差），有时还包括高阶矩，如偏度和峰度。但是，这些技术有一个严重的缺点，即如果分布中哪怕有几个偏离较远的点，结果也很容易失真。

> 在 Java 性能中，离群点代表交易缓慢和客户不满意。我们需要特别关注这些点，避免使用会忽略其重要性的技术。

从另一个角度考虑：除非已有大量的客户在抱怨，否则不大可能将改进平均响应时间作为目标。当然，这样做可以改善个人体验，但针对客户的不满意情况，我们会进行优化延迟。这意味着与正在接受满意服务的大多数人的体验相比，我们更应该关注离群事件。

图 5-6 显示了一个更为现实的曲线，说明方法（或交易）调用次数的可能性分布。这显然不是一个正态分布。

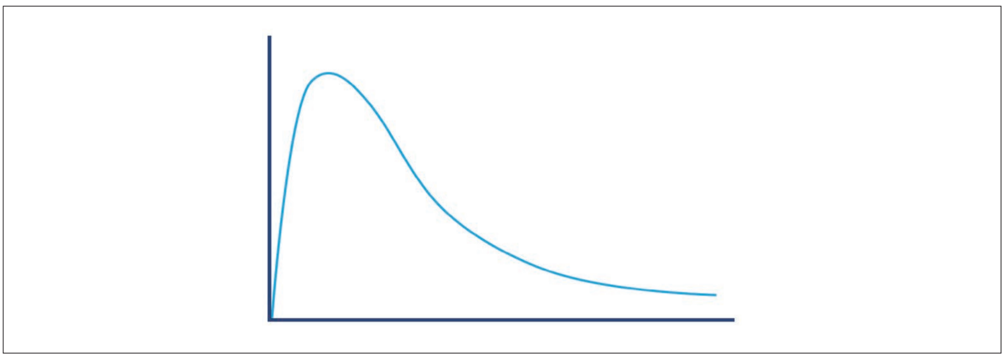


图 5-6：一个更现实的交易次数分布图

从图 5-6 中这个分布的形状可以看出一些我们对 JVM 已有的直观认识：它存在“热路径”，在该路径上，所有的相关代码都已经是 JIT 编译的，没有垃圾收集周期，等等。它们代表的是最佳情况（尽管也是一种常见情况），不会有哪个调用会随随便便就变快。

这违反了高斯统计的一个基本假设，使我们不得不考虑非正态分布。

> 非正态分布并不遵循正态分布统计的很多“基本规则”，对标准差 / 方差和其他高阶矩基本上也没用。

有一种技术对于处理 JVM 产生的非正态和“长尾”分布非常有用，那就是使用修改后的百分位数方案。不要忘了，分布是一个整体的图形，它用形状而不是单个数字来表示数据。

平均值试图用一个结果来表示整体分布情况，不同于仅计算平均值，我们可以按照时间间隔对分布进行采样。当用于正态分布的数据时，通常以固定间隔进行采样。不过只要稍加修改，我们就可以将这种技术应用于 JVM 统计。

修改的方法是，使用的采样要将长尾分布考虑进去，从平均值开始，然后是第 90 百分位数，并用对数校正，如下面的方法计时结果所示。这意味着我们是在按照一种更符合数据形状的模式进行采样。

```
50.0% level was 23 ns
90.0% level was 30 ns
99.0% level was 43 ns
99.9% level was 164 ns
99.99% level was 248 ns
99.999% level was 3,458 ns
99.9999% level was 17,463 ns
```

采样表明，尽管执行一个 getter 方法的平均时间是 23 纳秒，但对于 1000 个请求中的一个，时间就差了一个数量级；对于 100 000 个请求中的一个，则比平均值差了**两个**数量级。

长尾分布也可以称为**高动态范围分布**（high dynamic range distribution）。观测值的动态范围通常定义为最大记录值除以最小值（假设它不为零）。

将对数进行百分位数是理解长尾的一个有用的简单工具。然而，要进行更复杂的分析，我们可以使用一个公共库来处理高动态范围的数据集。这个库叫作 HdrHistogram，可以从 GitHub 上获得。它最初是由 Azul Systems 公司的 Gil Tene 创建的，Mike Barker、Darach Ennis 和 Coda Hale 也做了额外的工作。

> 直方图通过使用一组有限的范围区间（称为“桶”）来总结数据，并显示数据落入每个桶的频率。

HdrHistogram 也可以在 Maven 中央仓库上使用。在本书撰写时，它的当前版本是 2.1.9，要将其添加到自己的项目中，可以通过在 pom.xml 中添加如下依赖信息来实现：

```
<dependency>
    <groupId>org.hdrhistogram</groupId>
    <artifactId>HdrHistogram</artifactId>
    <version>2.1.9</version>
</dependency>
```

下面来看一个使用 HdrHistogram 的简单例子，这个例子接收一个由数字组成的文件，为连续结果之间的差值计算 HdrHistogram：

```
public class BenchmarkWithHdrHistogram {
    private static final long NORMALIZER = 1_000_000;

    private static final Histogram HISTOGRAM
            = new Histogram(TimeUnit.MINUTES.toMicros(1), 2);

    public static void main(String[] args) throws Exception {
        final List<String> values = Files.readAllLines(Paths.get(args[0]));
```

```
            double last = 0;
            for (final String tVal : values) {
                double parsed = Double.parseDouble(tVal);
                double gcInterval = parsed - last;
                last = parsed;
                HISTOGRAM.recordValue((long)(gcInterval * NORMALIZER));
            }
            HISTOGRAM.outputPercentileDistribution(System.out, 1000.0);
        }
    }
```

输出显示的是连续进行垃圾收集的时间。我们在第 6 章和第 8 章会看到，因为垃圾收集并不是以固定的时间间隔发生的，所以了解其频率分布很有用。下面是这个直方图绘制程序为一个垃圾收集日志样本产生的输出：

```
            Value     Percentile TotalCount 1/(1-Percentile)

            14.02 0.000000000000          1           1.00
          1245.18 0.100000000000         37           1.11
          1949.70 0.200000000000         82           1.25
          1966.08 0.300000000000        126           1.43
          1982.46 0.400000000000        157           1.67

    ...

         28180.48 0.996484375000        368         284.44
         28180.48 0.996875000000        368         320.00
         28180.48 0.997265625000        368         365.71
         36438.02 0.997656250000        369         426.67
         36438.02 1.000000000000        369
    #[Mean    =      2715.12, StdDeviation   =      2875.87]
    #[Max     =     36438.02, Total count    =          369]
    #[Buckets =           19, SubBuckets     =          256]
```

格式化器的原始输出很难分析，但幸运的是，HdrHistogram 项目包含了一个在线格式化器，可以用来将原始输出生成可视化的直方图。

对于这个例子，它产生的输出如图 5-7 所示。

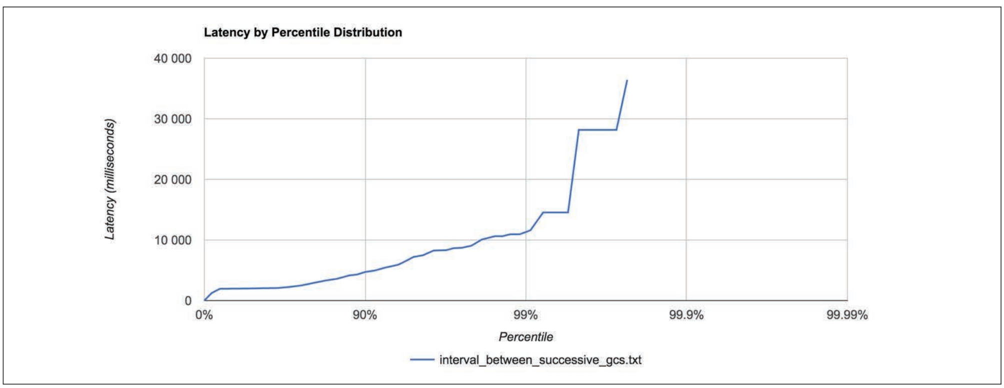


图 5-7：HdrHistogram 可视化示例

对于许多我们希望在 Java 性能调优中测量的观测值，统计数据往往是高度非正态的，而 HdrHistogram 是一个非常有用的工具，可以用可视化的方式将数据的形状显示出来以帮助我们理解。

5.4　统计的解释

经验数据和观察到的结果并不是一成不变的，而一个最难的、也是最为常见的工作就是解释我们从测量应用程序中得到的结果。

> 无论他们告诉你什么，都是人的问题。
>
> ——Gerald Weinberg

图 5-8 是一个真实的 Java 应用程序的内存分配率示例，该示例适用于性能良好的应用程序。截图来自于 Censum 垃圾收集分析器，我们将在第 8 章介绍。

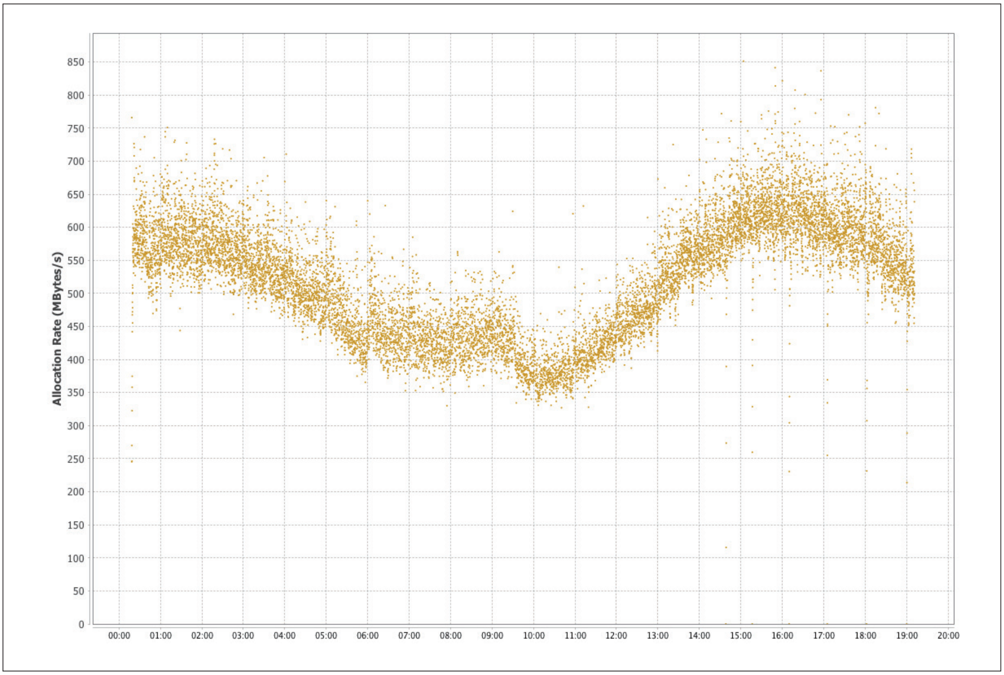


图 5-8：分配率示例

对分配数据的解释相对简单，因为存在着明显的信号。在覆盖的时间段内（几乎是一天），分配率基本稳定在每秒 350~700 MB。在 JVM 启动约 5 小时后分配率开始出现一个下降趋势，在 9~10 小时有一个明显的最小值，之后开始回升。

对可观测性数据进行解释的这类趋势非常普遍，因为分配率通常会反映出应用程序的实际工作量，而这将因一天中时间的不同而有很大的差异。然而，当我们解释真实的可观测数据时，情况会迅速变得更加复杂。这就导致有时会出现所谓的“帽子 / 大象”问题，该问

题出自 Antoine de Saint-Exupéry 的著作《小王子》。

图 5-9 说明了该问题。我们最初只能看到一个复杂的 HTTP 请求 – 响应时间的直方图。然而，就像书中的叙述者一样，如果我们能够进一步想象或者分析，就会发现这个复杂的画面其实是由几个相当简单的部分组成的。

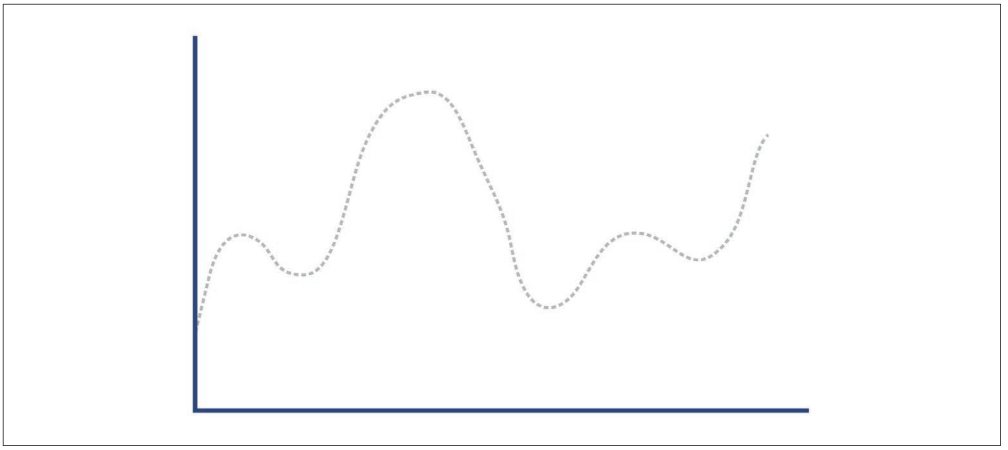


图 5-9：帽子，还是大象被蟒蛇吃了

解码响应直方图的关键是要认识到"Web 应用程序响应"是一个非常笼统的类别，包括成功的请求（所谓的 2xx 响应）、客户端错误（4xx，包括常见的 404 错误）和服务器错误（5xx，特别是内部服务器错误 500）。

每种响应类型对于响应时间都有不同的特征分布。如果客户端提出的 URL 没有映射的请求（404），则 Web 服务器可以立即回复响应。这意味着仅客户端错误响应的直方图看起来更像图 5-10。

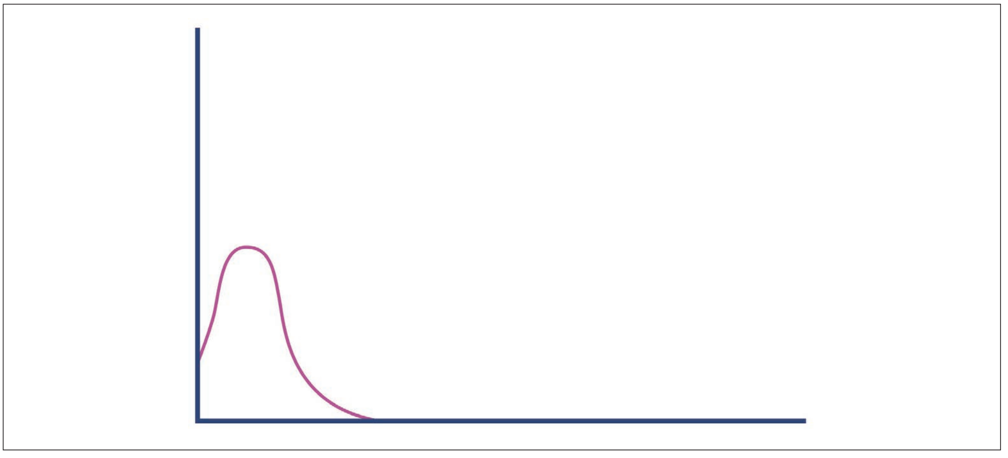


图 5-10：客户端错误

相比之下，服务器错误往往在大量的处理时间被消耗之后才会发生（例如，由于后台资源紧张或超时）。因此，服务器错误响应的直方图可能类似于图 5-11。

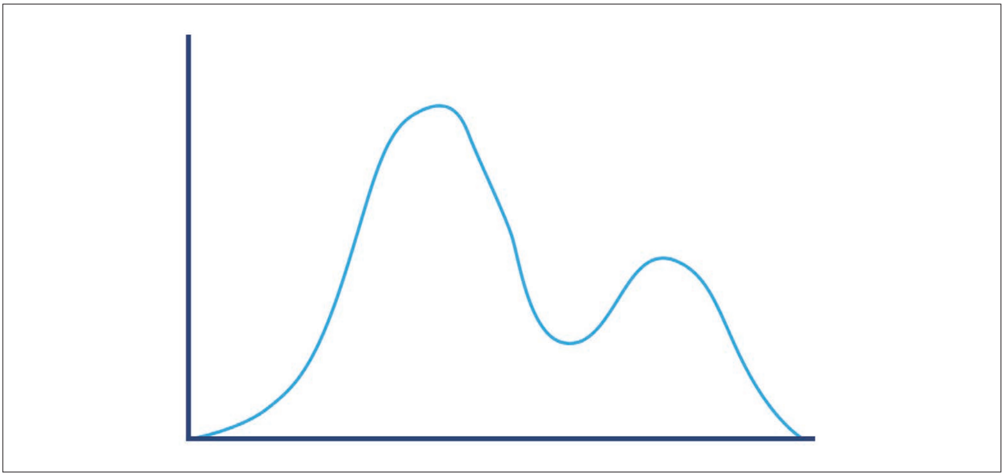


图 5-11：服务器错误

成功的请求会有一个长尾分布，但实际上，我们可能期望响应分布是“多模态”的，并且有几个局部的最大值。如图 5-12 所示，它代表了可能有两条共同的执行路径通过该应用程序，但响应时间截然不同。

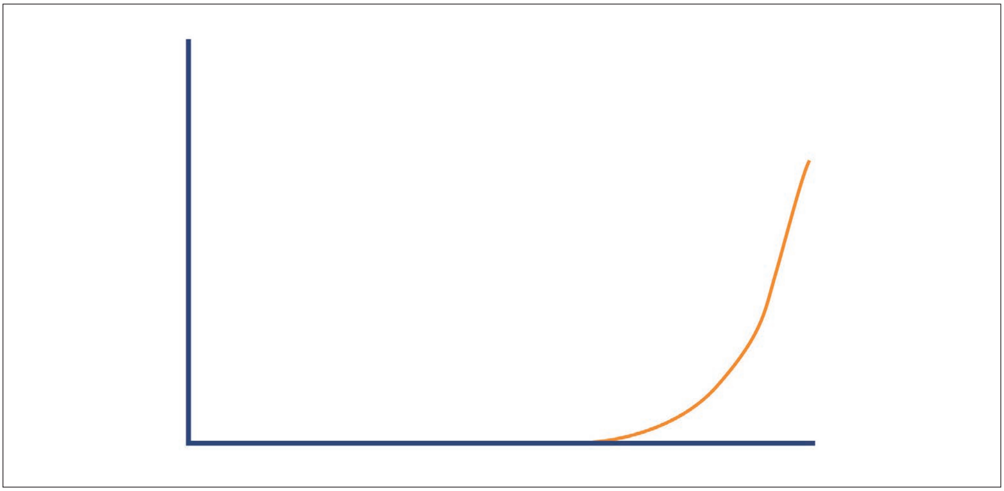


图 5-12：成功的请求

将这些不同类型的响应组合成一个图，将得到如图 5-13 所示的结构。我们已经从 separate 直方图中重新得到了原来的“帽子”形状。

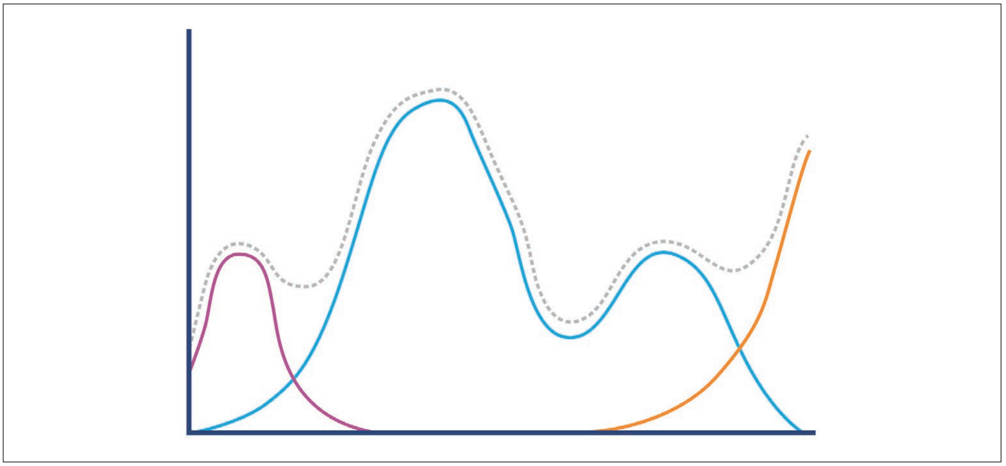


图 5-13：重温帽子或大象

将一般的可观测数据分解成更有意义的子群是一个非常有用的概念，它表明在尝试从结果中推断出结论之前，需要确保我们足够了解数据和领域。我们很可能希望将数据进一步分解成更小的数据集，例如，成功的请求在读取的请求中相对于更新或上传的请求来说会有非常不同的分布。

PayPal 的工程团队已经写了很多关于他们使用统计和分析的文章。他们有一个博客，里面有很好的资源。特别是，Mahmoud Hashemi 撰写的“Statistics for Software”一文很好地介绍了他们的方法论，其中包括前面讨论的有关帽子 / 大象这个问题的内容。

5.5　小结

微基准测试是 Java 性能最接近于“暗黑艺术”的地方。虽然这种描述令人回味，但也不是完全贴切。它仍然是一个由在职开发人员所承担的工程学科。在使用微基准测试之前，请考虑以下内容。

- 除非知道是一个已知用例，否则不要使用微基准测试。
- 如果必须使用微基准测试，请使用 JMH。
- 尽可能公开地讨论你的结果，并与同行交流。
- 做好经常犯错的准备，不断挑战自己的思维。

微基准测试的一个好处是它揭示了由低级子系统产生的高度动态行为和非正态分布，从而可以帮助我们更好地理解 JVM 的复杂性和形成思维模型。

下一章将从方法论入手，开始对 JVM 的内部结构和主要子系统进行技术上的深入研究，首先是对垃圾收集的研究。

第6章

理解垃圾收集

Java 环境有几个标志性或定义性特性，而垃圾收集（garbage collection，GC）就是其中最典型的一个。不过在 Java 平台发布之初，很多人对垃圾收集怀有敌意。火上浇油的是，Java 还有意没有提供任何语言级别的方法来控制收集器的行为（后续的版本也没有）。[1]

这意味着在早期，人们对 Java 垃圾收集的性能存在一定程度的不满，而这也成为人们对整个 Java 平台的看法。

然而，对强制式、用户不可控的垃圾收集的早期设想，目前已经得到充分证明；现在已经很少有程序员还坚持认为内存应该手动管理了。即便现在，人们对系统编程语言（比如 Go 和 Rust）的看法也是把内存管理当作编译器和运行时，而不是程序员应该考虑的问题（除了一些特殊情况）。

Java 垃圾收集的本质是，与其要求程序员理解系统中每个对象精确的生命周期，不如让运行时代替程序员记录对象信息，并在不再需要这些对象时自动将其释放。自动回收的内存之后可被清理和复用。

垃圾收集的所有实现都必须遵循下面两个基本规则：

- 算法必须回收所有垃圾；
- 不能回收任何活跃对象。

两个规则中，第二个更重要。回收活跃对象会导致段错误（segmentation fault），甚至更糟糕的是，可能会默默破坏程序数据。Java 的垃圾收集算法需要确保它们不会回收程序仍在使用的对象。

程序员不用手动管理每个底层细节，为此，他们必须放弃一些底层控制。这种理念是

注 1：虽然有 `System.gc()` 方法，但基本派不上什么用场。

Java 的托管方式的本质所在。这也体现了 James Gosling 把 Java 当作完成工作的蓝领语言的理念。

本章将介绍 Java 垃圾收集的一些基本理论，并解释为什么 Java 垃圾收集是平台中最难理解和控制的部分之一。我们还会介绍 HotSpot 运行时系统的基本特性，包括诸如它在运行的时候如何表示堆中的对象等细节。

本章的最后还会介绍 HotSpot 中最简单的生产级垃圾收集器——并行收集器，并解释它能适合多种负载的原因。

6.1 标记和清除

如果被追问的话，大部分 Java 程序员能回想起 Java 的垃圾收集依赖于一个叫**标记和清除**（mark and sweep）的算法，但大多数人很难记起有关实际操作过程的任何细节。

本节将介绍该算法的一种基本形式，并演示如何使用它自动回收内存。这是一种经过简化的算法形式，只是为了介绍一些基本概念，并不代表生产级 JVM 实际如何执行垃圾收集。

标记和清除算法的这一介绍性形式使用了一个已分配对象链表（allocated object list）来保存指向每个已经分配但尚未回收的对象的指针。整个垃圾收集算法描述如下。

1. 循环遍历已分配链表，清空标记位。
2. 从 GC 根开始，寻找活跃对象。
3. 在到达的每个对象上设置一个标记位。
4. 循环遍历已分配链表，对于每个标记位尚未设置的对象。
   a. 回收堆中内存，将其放回空闲链表。
   b. 从已分配链表中移除该对象。

活跃对象通常是通过深度优先搜索来定位的，生成的对象图叫作**活跃对象图**（live object graph）。有时也叫作**可达对象的传递闭包**（transitive closure of reachable object），如图 6-1 所示。

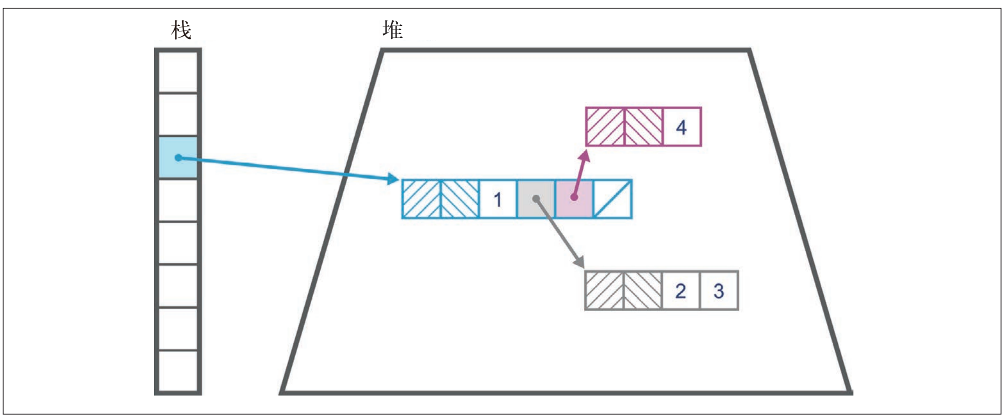


图 6-1：内存布局的简单视图

堆的状态很难可视化地表现出来，幸运的是，可以借助一些简单的工具。最简单的是 `jmap -histo` 命令行工具。它会显示每个类型分配的字节数，以及共同占用了这些内存的实例的数量。它的输出类似下面这样：

```
num    #instances       #bytes  class name
----------------------------------------------
  1:        20839     14983608  [B
  2:       118743     12370760  [C
  3:        14528      9385360  [I
  4:          282      6461584  [D
  5:       115231      3687392  java.util.HashMap$Node
  6:       102237      2453688  java.lang.String
  7:        68388      2188416  java.util.Hashtable$Entry
  8:         8708      1764328  [Ljava.util.HashMap$Node;
  9:        39047      1561880  jdk.nashorn.internal.runtime.CompiledFunction
 10:        23688      1516032  com.mysql.jdbc.Co...$BooleanConnectionProperty
 11:        24217      1356152  jdk.nashorn.internal.runtime.ScriptFunction
 12:        27344      1301896  [Ljava.lang.Object;
 13:        10040      1107896  java.lang.Class
 14:        44090      1058160  java.util.LinkedList$Node
 15:        29375       940000  java.util.LinkedList
 16:        25944       830208  jdk.nashorn.interna...FinalScriptFunctionData
 17:           20       655680  [Lscala.concurrent.forkjoin.ForkJoinTask;
 18:        19943       638176  java.util.concurrent.ConcurrentHashMap$Node
 19:          730       614744  [Ljava.util.Hashtable$Entry;
 20:        24022       578560  [Ljava.lang.Class;
```

还有一个可以使用的 GUI 工具：第 2 章介绍过的 VisualVM 的 Sampling 页。VisualVM 的 VisualGC 插件提供了一个堆变化情况的实时视图，但是一般而言，从一个时刻到另一个时刻的堆视图还不足以用于精确分析，要获得更好的结果，应该使用垃圾收集日志，第 8 章会重点介绍。

垃圾收集术语

用于描述垃圾收集算法的术语有点混乱，有些术语的意义还会随着时间的推移发生变化。为清晰起见，下面提供了一个基本的术语表，说明本书是如何使用具体术语的。

全部停顿（stop-the-world，STW）

  在垃圾收集时，垃圾收集周期要求所有的应用程序线程停顿。这可以避免在垃圾收集时，应用程序代码破坏垃圾收集线程所掌握的堆状态信息。大部分简单的垃圾收集算法都是如此。

并发（concurrent）

  垃圾收集线程可以在应用程序线程运行的时候运行。实现并发十分困难，而且计算成本也非常高。几乎没有算法是真正并发的。而且，我们需要使用复杂的技巧来挖掘并发收集的优势。在 7.3 节中我们会看到 HotSpot 中的并发标记和清除（concurrent mark and sweep，CMS）收集器，名虽如此，或许用“基本并发”收集器来描述更为恰当。

并行（parallel）

  有多个线程用于执行垃圾收集。

精确（exact）

对于堆的状态，精确垃圾收集模式有足够的类型信息，从而确保所有垃圾都能在一个周期内完成回收。更宽泛地说，精确模式拥有区分一个 `int` 和一个指针所需的属性（property）。

保守（conservative）

保守模式缺乏精确模式所具备的信息。因此，保守模式经常会浪费资源，而且因为它们根本不了解自己所代表的类型系统，所以往往不那么高效。

移动（moving）

在移动垃圾收集器中，对象在内存中的位置有可能变化，因此它们的地址不是稳定的。支持使用原始指针的环境（如 C++）不是特别适合移动收集器。

压缩（compacting）

在收集周期结束时，已分配的内存（即存活的对象）被组织为一个单一的连续区域（通常在该区域的开始处），并有一个指针用来指示可供写入对象的空闲空间的起始位置。压缩收集器可以避免内存碎片。

疏散（evacuating）

在收集周期结束时，已收集区域完全为空，所有的活对象都被移动（或疏散）到另一个内存区域。

在大多数其他语言和环境中，也使用了同样的术语。比如，Firefox Web 浏览器（SpiderMonkey）中的 JavaScript 运行时也使用了垃圾收集，并且近年来也在加入目前 Java 的垃圾收集实现中已经存在的一些特性（如精确性和压缩）。

6.2 HotSpot运行时

除了一般的垃圾收集术语外，HotSpot 还引入了一些专属于自身实现的术语。为了全面了解垃圾收集在这款 JVM 上是如何工作的，需要掌握 HotSpot 内部的一些细节。

为了学习接下来的内容，记住 Java 只有两种类型的值是非常有帮助的：

- 基本类型（`byte`、`int` 等）；
- 对象引用。

许多 Java 程序员会很宽泛地谈论**对象**，但就我们的目的而言，记住这一点非常重要：与 C++ 不同，Java 没有通用的地址解引用（address dereference）机制，只能使用一个**偏移操作符**（`.`）来访问字段和调用**对象引用**的方法。另外还请记住，Java 的方法调用语义是纯粹的按值调用（call-by-value），尽管对于对象引用来说，这意味着被复制的值是对象在堆中的地址。

6.2.1 对象的运行时表示

HotSpot 通过一个叫作 `oop` 的结构来表示运行时 Java 对象。这是**普通对象指针**（ordinary object pointer）的简称，是 C 语言意义上的真正指针。这些指针可以放置在引用类型的局

部变量中，在该变量中它们从 Java 方法的栈帧指向包含 Java 堆的内存区域。

有几种不同的数据结构组成了 oop 家族，而用于表示 Java 类的实例的类型叫作 instanceOop。

instanceOop 的内存布局是从每个对象上都有的两个机器字的头部开始的。其中的第一个是 mark word，它是一个指针，指向特定于该实例的元数据。下一个是 klass word，它指向类级别的元数据。

在 Java 7 和之前的版本中，instanceOop 的 klass word 指向一个名为 PermGen 的内存区域，它是 Java 堆的一部分。一般的规则是，Java 堆中的任何东西都必须有一个对象头。在这些旧的 Java 版本中，我们把元数据称为 klassOop。klassOop 的内存布局很简单，就是对象头后面紧跟着 klass 元数据。

从 Java 8 开始，klass 被保存在 Java 堆的主要部分之外（但不在 JVM 进程的 C 堆之外）。在这些 Java 版本中，klass 字不需要对象头，因为它们指向 Java 堆之外。

开头的 k 是用来帮助区分 klassOop 和代表 Java Class<?> 对象的 instanceOop，它们不是一回事。

在图 6-2 中，我们可以看到两者的区别。从根本上说，klassOop 包含对应类的虚函数表（vtable），而 Class 对象包含了一个指向 Method 对象的引用数组，用于反射式调用。在第 9 章讨论 JIT 编译时，会有更多相关讨论。

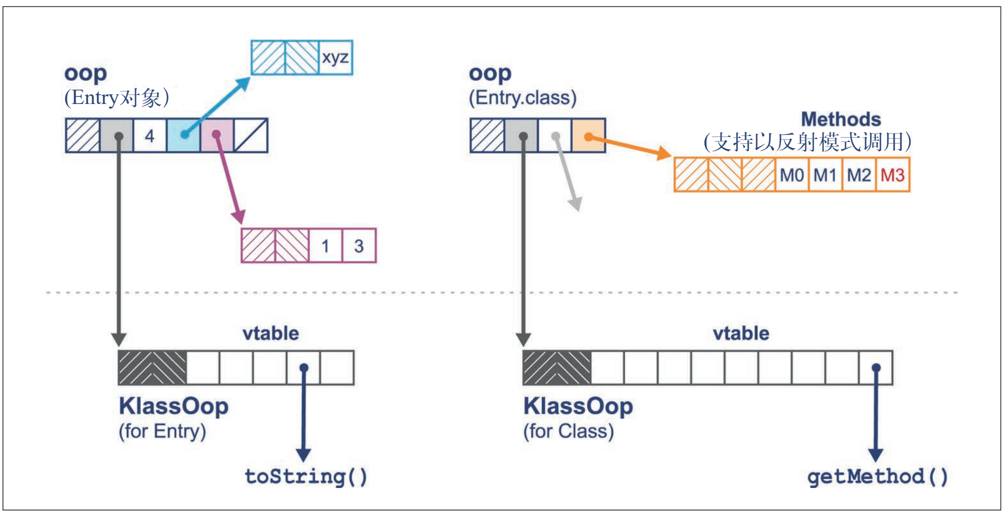


图 6-2：klassOop 和 Class 对象

oop 通常是机器字，所以在传统的 32 位机器上是 32 位，而在现代处理器上是 64 位。然而，这可能会浪费大量的内存。为了缓解这种情况，HotSpot 提供了一种叫作**压缩指针**（compressed oops）的技术。如果设置了 -XX:+UseCompressedOops 这个选项（在 Java 7 及以

后的版本中，在 64 位机器上该选项默认开启），则堆中的如下 oop 会被压缩：

- 堆中每个对象的 klass word
- 引用类型的实例字段
- 对象数组的每一个元素

一般而言，这意味着一个 HotSpot 对象的头部由以下部分组成：

- mark word（完整大小）
- klass word（可能被压缩）
- length word（如果对象是一个数组），它总是 32 位的
- 一个 32 位大小的间隙（如果对齐规则有要求的话）

对象的实例字段紧跟在头部之后。对于 `klassOop`，方法的虚函数表直接跟在 klass word 后面。压缩后的 oop 的内存布局如图 6-3 所示。

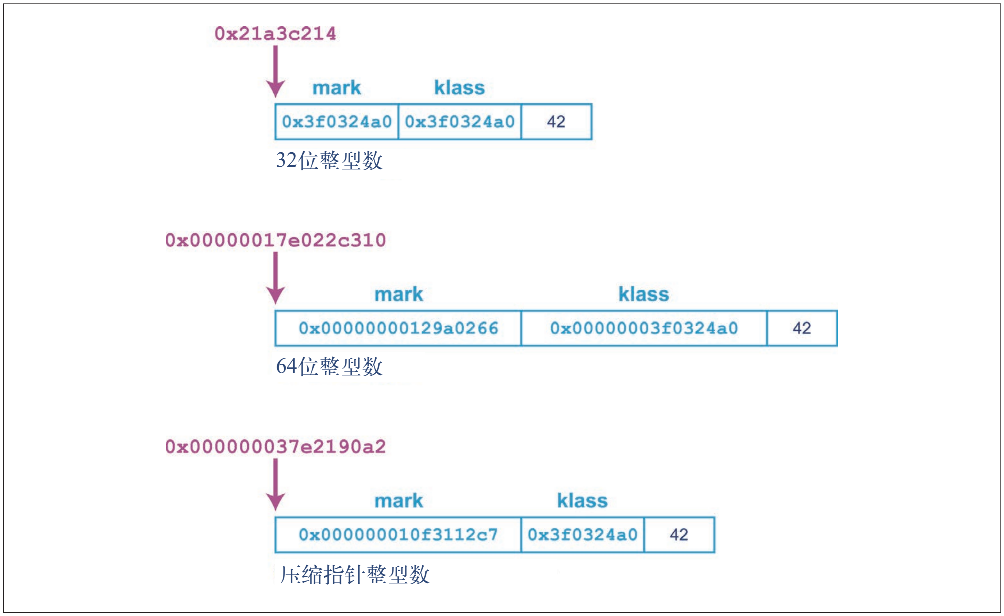


图 6-3：压缩的 oop

过去，一些对延迟极为敏感的应用程序偶尔可以通过关闭压缩指针特性来提高性能，但代价是增加了堆的大小（通常会增加 10%~50%）。然而，能够获得可测量的性能优势的应用程序种类其实非常少，对于大多数现代应用程序来说，这反而是第 4 章中反模式“摆弄开关”的典型例子。

正如我们所了解的 Java 基础知识，数组是对象。这意味着 JVM 的数组也是以 oop 的形式来表示的。这就是为什么数组的元数据既有第 3 个字，也有通常的 mark word 和 klass word。第 3 个字是数组的长度，这也解释了为什么 Java 中的数组索引被限制为 32 位值。

> 使用额外的元数据来携带数组的长度缓解了 C 和 C++ 中存在的一系列问题；在 C 和 C++ 中，如果不知道数组的长度，则意味着必须向函数传递额外的参数。

除了指向 `instanceOop`（或 `null`），JVM 托管环境不允许 Java 引用指向其他任何地方。这意味着在底层：

- 一个 Java 值是一个比特模式，要么对应一个基本类型的值，要么对应一个 `instanceOop`（引用）的地址；
- 任何被视为指针的 Java 引用均指向 Java 堆主体部分中的一个地址；
- 作为 Java 引用的目标的地址中，均包含一个 mark word，下一个机器字是 klass word；
- 一个 `klassOop` 和一个 `Class<?>` 的实例是不同的（因为前者存在于堆的元数据区域中），而且一个 `klassOop` 是不能放在 Java 变量中的。

HotSpot 在一系列 .hpp 头文件中定义了 oop 的层次结构，这些文件被保存在 OpenJDK 8 源码树的 hotspot/src/share/vm/oops 中。oop 的整体继承层次结构是这样的：

```
oop（抽象基类）
 instanceOop（实例对象）
 methodOop（方法的表示）
 arrayOop（数组抽象基类）
 symbolOop（内部符号/字符串类）
 klassOop（klass头部，只存在于Java 7和更早的版本中）
 markOop
```

使用 oop 结构来表示运行时的对象，用一个指针来指向类级的元数据，另一个指针来指向实例级的元数据，这种用法并不少见。其他许多 JVM 和执行环境也使用了相关机制。比如，Apple 的 iOS 就使用了类似的模式来表示对象。

6.2.2　GC根和Arena

关于 HotSpot 的文章和博客经常引用 **GC 根**。这些是内存的“锚点”，本质上是已知的指针，处于它感兴趣的内存池之外，并指向这个内存池。它们是**外部**指针，和**内部**指针截然不同。内部指针是处于一个内存池内，并指向当前内存池内的另一个位置。

图 6-1 中是一个 GC 根的示例。不过我们还将看到其他类型的 GC 根，包括：

- 栈帧
- JNI
- 寄存器（用于变量提升情况）
- 代码根（来自 JVM 代码缓存）
- 全局变量
- 来自已加载类的元数据

如果感觉这个定义比较复杂，那么最简单的 GC 根的例子就是引用类型的局部变量，它总是指向堆中的一个对象（假设它不为 `null`）。

HotSpot 垃圾收集器是以内存区域的方式工作的，这里的内存区域被称为 Arena。这是一个非常底层的机制，Java 开发人员通常不需要这么详细地考虑内存系统的运行机制。然而，性能专家有时需要深入研究 JVM 的内部，因此熟悉相关文献中使用的概念和术语是有帮助的。

有一个重要的事实需要记住，即 HotSpot 不使用系统调用来管理 Java 堆。相反，正如 3.6 节中所讨论的，HotSpot 通过用户空间代码来管理堆的大小，所以我们可以使用简单的观测对象来判断垃圾收集子系统是否正在引发某些类型的性能问题。

下一节将仔细研究驱动任何 Java 或 JVM 工作负载的垃圾收集行为的两个最重要的特性。对于任何想要真正掌握驱动 Java 垃圾收集（Java 垃圾收集是 Java 性能的主要驱动因素之一）的因素的开发人员来说，充分理解这些特性是必不可少的。

6.3 分配与生命周期

一个 Java 应用程序的垃圾收集行为主要有两个驱动因素：

- 分配率
- 对象生命周期

分配率是新创建的对象在一个时间段内所使用的内存量（通常以 MB/s 为单位）。它并不是由 JVM 直接记录下来的，但这个值很容易估算，并且也可以使用 Censum 这样的工具来精确测定。

相比之下，对象的生命周期通常很难测量（甚至估算）。事实上，反对使用手动内存管理的主要论据之一，就是要真正理解实际应用程序中的对象生命周期过于复杂。因此，对象生命周期甚至比分配率更重要。

> 垃圾收集也可以被认为是“内存回收和再利用”。因为对象寿命短暂，所以能够反复使用同一块物理内存是垃圾收集技术的一个关键假设。

对象被创建出来，存活一段时间，之后用于存储其状态的内存可以被回收，这个理念至关重要；没有这个理念，垃圾收集将无法工作。正如我们将在第 7 章中看到的，垃圾收集器必须多方权衡，而其中最重要的一些权衡就是由生命周期和分配问题所驱动的。

弱分代假说

JVM 内存管理的一个关键部分依赖于一个观察到的软件系统在运行时的表现，即弱分代假说（weak generational hypothesis）：

> 在 JVM 和类似的软件系统中，对象生命周期的表现为双峰分布——大部分对象寿命很短，次一级对象的寿命长于预期。

这个假说实际上是从实验中观察到的关于面向对象工作负载行为的一个经验法则，由此得

出一个明显的结论：垃圾收集堆应该以这样的方式组织，即能够方便快速地收集寿命短的对象，并且最好让寿命长的对象可以与寿命短的隔离开来。

HotSpot 使用了下面几种机制来尝试利用弱分代假说：

- 它会跟踪每个对象的“年龄”（或者说代数，即该对象到目前为止熬过的垃圾收集次数）；
- 除了大对象外，它在“Eden”空间（也叫 Nursery）中创建新对象，并且期望移动存活的对象；
- 它维护着一个单独的内存区域（Old Generation 或者 Tenured Generation）来保存那些已经存活足够长而且很有可能继续存活下去的对象。

这种方法引出了如图 6-4 所示的简化形式的视图，其中存活了一定数量的垃圾收集周期的对象被提升到了老年代。注意区域是连续的。

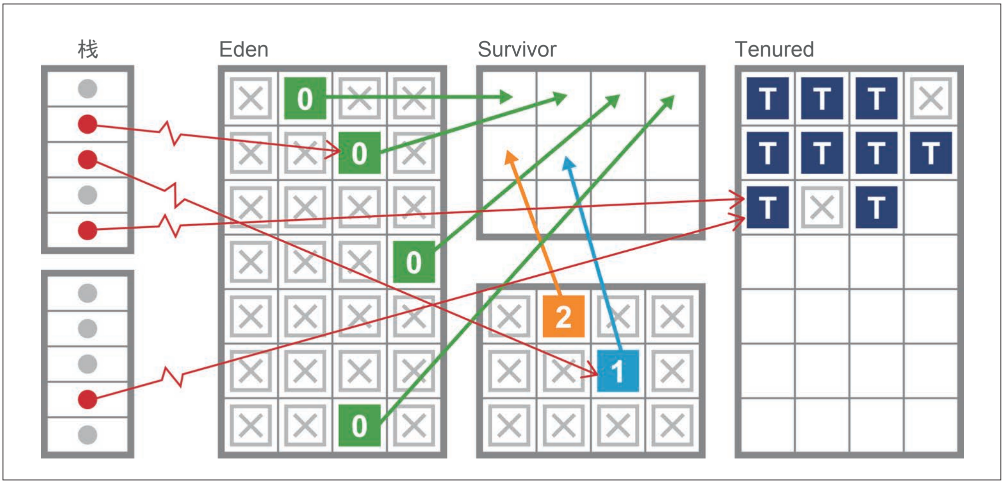


图 6-4：分代收集

为了进行分代收集而将内存划分成不同区域，会给 HotSpot 如何实现标记和清除收集带来额外的影响。一项重要的技术涉及跟踪记录从外部指向新生代内的指针。这使得垃圾收集周期不必遍历整个对象图来确定仍然活着的新生代对象。



“有极少数从老年代指向新生代对象的引用”，这句话有时被引用作为弱分代假说的第二部分。

为方便处理，HotSpot 维护了一个叫作**卡表**（card table）的结构，以帮助记录哪些老年代对象可能指向新生代对象。卡表本质上是一个由 JVM 管理的字节数组，该数组的每个元素对应于老年代中一个 512 字节的区域。

其核心思想是，当老年代中对象 `o` 中的一个引用类型的字段被修改时，包含与 `o` 相对应的 `instanceOop` 的卡表项被标记为“脏”（dirty）。当更新引用字段时，HotSpot 通过一个简单

的**写屏障**（write barrier）来实现这一点。它本质上可以归结为在字段存储后执行了这样一段代码：

```
cards[*instanceOop >> 9] = 1;
```

注意卡片的脏值为 `1`，右移 9 位后得到卡表的大小为 512 字节。

最后，需要注意的是，在历史上，用新生代和老年代来描述堆是 Java 收集器管理内存的方式。随着 Java 8u40 的到来，一种新的收集器（garbage first，G1）达到了生产质量。G1 选择了不同的堆布局方法，这将在 7.4 节中介绍。关于堆管理的这种新的思维方式将变得越来越重要，因为 Oracle 的意图是让 G1 成为 Java 9 以后的默认收集器。

6.4　HotSpot中的垃圾收集

回顾一下，与 C/C++ 和类似环境不同，Java 不使用操作系统来管理动态内存。相反，当 JVM 进程启动时，JVM 预先分配（或保留）内存，并在用户空间管理单一连续的内存池。

正如我们所看到的，这个内存池是由多个具有特定用途的不同区域组成的，而且对象的地址会经常发生变化。这是因为垃圾收集器会重新安置（relocate）通常在 Eden 区创建的对象。执行重新安置的收集器被称为“疏散”收集器，6.1 节曾提到过。HotSpot 可以使用的许多收集器是疏散式的。

6.4.1　线程本地分配

JVM 使用了一个性能增强来管理 Eden 区。这是一个需要进行高效管理的关键区域，因为大多数对象是在这里创建的，而且寿命非常短的对象（生命周期小于下一个垃圾收集周期剩余时间的那些）永远不会再出现在其他地方。

为了提高效率，JVM 将 Eden 区划分成若干个缓冲区，并将这些缓冲区交给应用程序线程使用，用于新对象的分配。这种方法的好处是，每个线程都知道自己不需要考虑其他线程在该缓冲区内分配的可能性。这些区域叫作**线程本地分配缓冲区**（thread-local allocation buffer，TLAB）。

> HotSpot 会动态设置分给应用程序线程的 TLAB 的大小，所以如果某个线程的内存就要耗尽了，可以为其分配更大的 TLAB，以减少向该线程提供缓冲区的开销。

应用程序线程对其 TLAB 的这种排他性控制，意味着对 JVM 线程来说，分配操作的时间复杂度是 $O(1)$。这是因为当一个线程创建一个新对象时，将为该对象分配存储空间，并且将线程本地指针更新为指向下一个空闲的内存地址。从 C 运行时角度看，这是一个简单的指针碰撞，也就是多了一条指令，将“next free”指针向前移动了一下。

这种行为如图 6-5 所示，其中每个应用程序线程都持有一个用来分配新对象的缓冲区。如果应用程序线程填满了其缓冲区，那么 JVM 会提供一个指向 Eden 区中的新区域的指针。

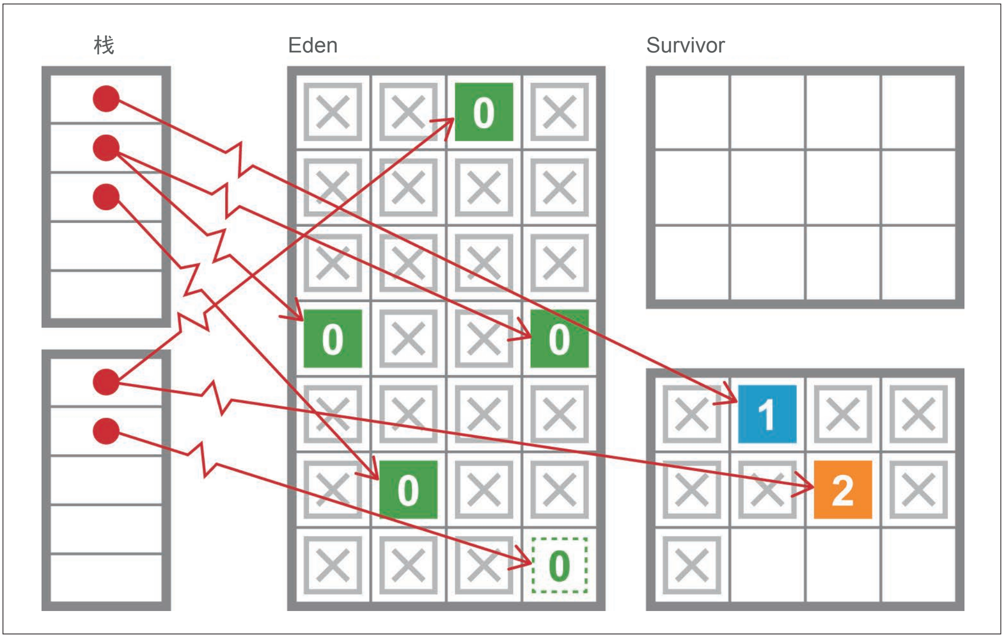


图 6-5：线程本地分配

6.4.2　半空间收集

疏散式收集器有种特殊情况值得注意。这种类型的收集器有时被称为**半空间疏散式收集器**（hemispheric evacuating collector），它会使用两个空间（通常大小相同）。其核心思想是将这些空间作为那些实际寿命比较短的对象的临时存储区。这样可以防止短寿命的对象把老年代弄乱，还可以降低整堆收集（Full GC）的频率。这些空间有两个基本属性：

- 当收集器正在收集当前活跃的半空间时，对象会被以压缩方式移动到另一半空间，完成收集的一半空间会被清空以等待复用；
- 任何时候，总有一半的空间是空的。

当然，这种方法使用的内存是收集器半空间部分实际可以容纳的内存的两倍。这在某种程度上是浪费，但如果空间不太大，这种技术往往就非常有用。HotSpot 就是利用这种半空间方法，结合 Eden 空间，为新生代提供了一个收集器。

HotSpot 新生代堆的半空间部分被称为 **Survivor 空间**。从图 6-6 所示的 VisualGC 视图中可以看到，与 Eden 相比，Survivor 空间通常比较小，而且它的角色会随着每一次新生代的垃圾收集而互换。第 8 章将讨论如何调整 Survivor 空间的大小。

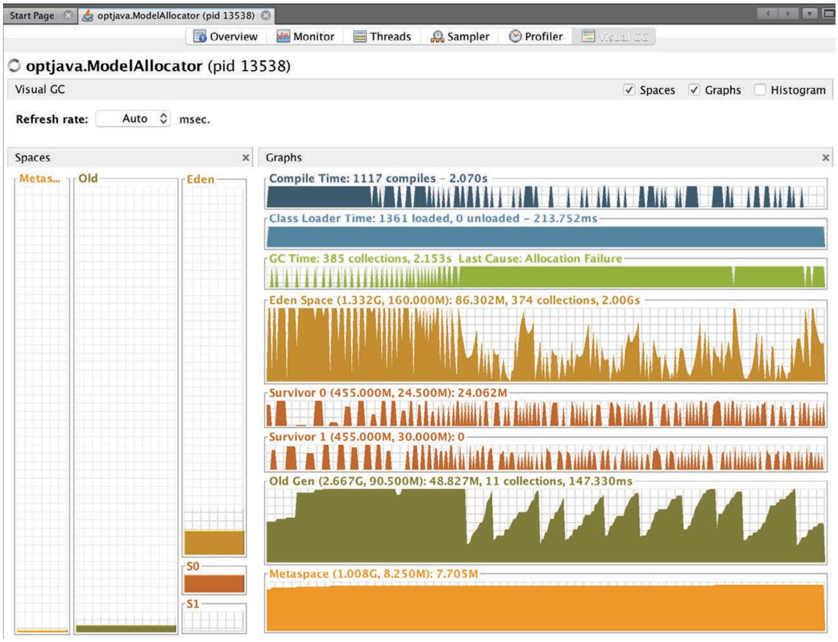


图 6-6：VisualGC 插件

我们在 2.7 节介绍了 VisualVM，它的 VisualGC 插件是一个非常有用的、初步的垃圾收集调试工具。正如第 7 章将讨论的，与 VisualGC 所使用的即时 JMX 数据相比，垃圾收集日志包含更多有用的信息，而且支持对垃圾收集进行更深入的分析。然而，在开始新的分析时，简单地了解应用程序的内存使用情况往往是很有帮助的。

使用 VisualGC 可以看到垃圾收集的一些聚合效果，例如对象在堆中被重新安置，以及每次新生代收集时发生在 Survivor 空间之间的循环交换。

6.5 并行收集器

在 Java 8 和更早的版本中，JVM 的默认收集器是并行收集器。它们的新生代收集和老年代收集都是全部停顿的（STW），而且针对吞吐量进行了优化。在停止所有的应用程序线程之后，并行收集器会使用所有可利用的 CPU 核心来尽快回收内存。可用的并行收集器如下。

Parallel GC

　用于新生代的最简单的收集器。

ParNew

　与 CMS 收集器配合使用的并行垃圾收集的变种。

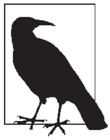
ParallelOld

　用于老年代的并行收集器。

这几个并行收集器彼此在某些方面有些相似——在设计上，它们都使用多个线程来尽快找到活对象，同时尽可能减少簿记信息。不过它们之间也有一些区别，所以下面我们来看看新生代收集和老年代收集这两种主要类型。

6.5.1 新生代并行收集

最常见的收集类型是新生代收集。它通常发生在这样的情况下：一个线程试图向 Eden 空间中分配一个对象，但在其 TLAB 中已经没有足够的空间，而且 JVM 也无法为该线程分配一个新的 TLAB。此时 JVM 除了停止所有的应用程序线程之外别无选择，因为如果一个线程无法分配，那么很快所有的线程就都将无法分配了。



线程也可以在 TLAB 之外进行分配（比如，对于大块内存）。当然在理想的情况下，非 TLAB 分配的比率不能过高。

一旦所有的应用程序线程（或者用户线程）停止，HotSpot 就会查看新生代（它被定义为 Eden 空间和当前非空的 Survivor 空间），并识别出所有非垃圾对象。它会将 GC 根作为并行标记扫描的起点（还会利用卡表来识别来自老年代的 GC 根）。

然后，Parallel GC 收集器会将所有幸存的对象疏散到当前为空的 Survivor 空间中（并在重新安置时增加这些对象的代数）。最后，将 Eden 和刚刚疏散完的 Survivor 空间标记为空的、可重复使用的空间，并启动应用程序线程，这样也就可以重新开始向应用程序线程分发 TLAB 了。这一过程如图 6-7 和图 6-8 所示。

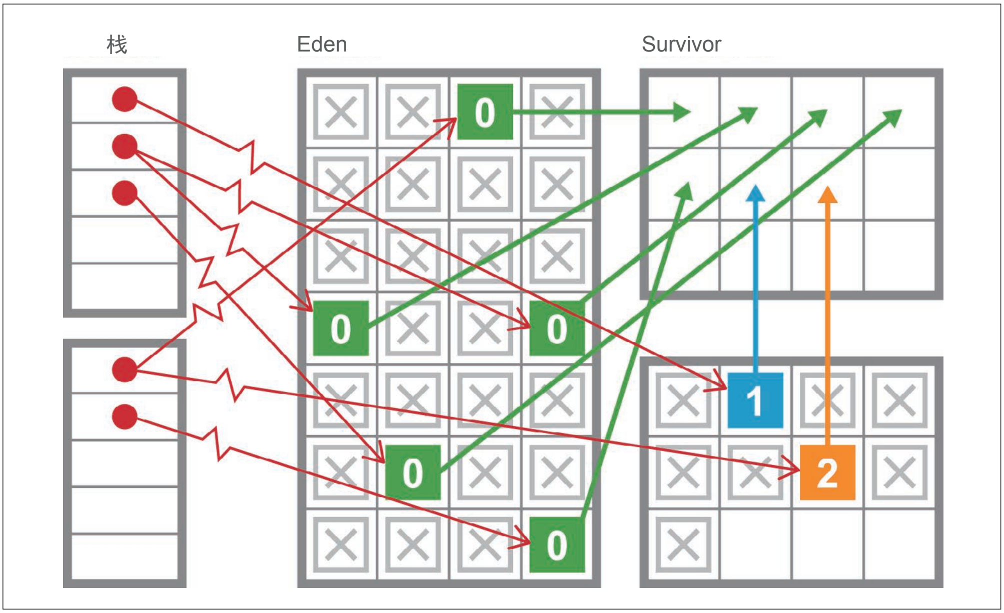


图 6-7：新生代收集

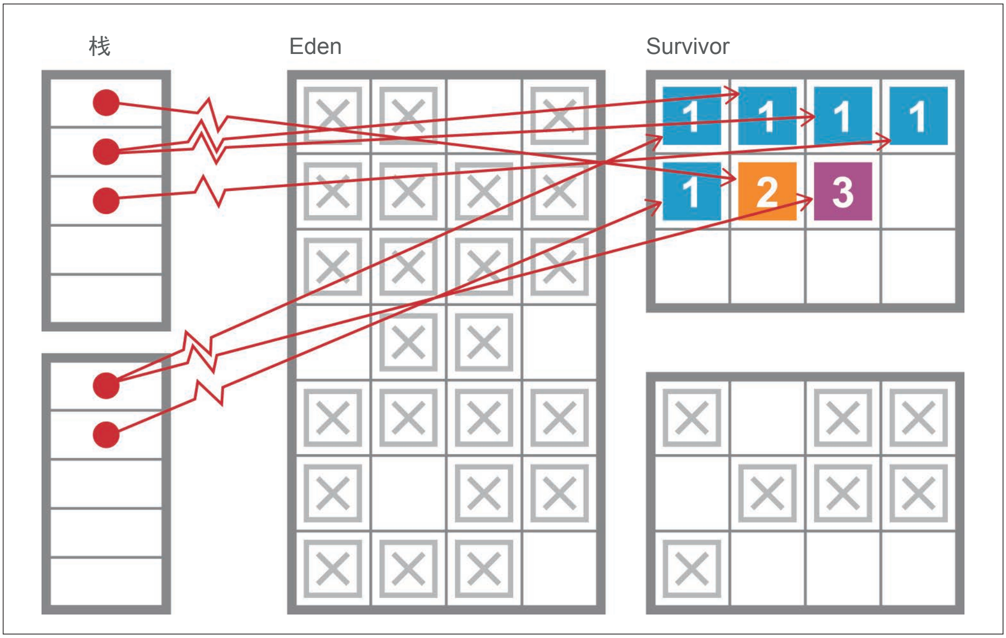


图 6-8：新生代疏散

这种方法试图通过只接触活对象来充分利用弱分代假说。它还希望尽可能高效，尽可能使用所有的 CPU 核心来运行，以缩短全部停顿的暂停时间。

6.5.2 老年代并行收集

ParallelOld 收集器是目前（从 Java 8 开始）默认的老年代收集器。它与 Parallel GC 在很多地方非常相似，但也有一些根本的区别。特别是，Parallel GC 是半空间疏散型收集器，而 ParallelOld 是只有一个连续内存空间的压缩型收集器。

这意味着，由于老年代没有其他空间可以疏散，因此 ParallelOld 收集器要尝试在老年代之内重新安置对象，以回收死亡对象留下的空间。因此，这种收集器可以非常高效地使用内存，而且不会受内存碎片的困扰。

这将产生非常高效的内存布局，但代价是在整个垃圾收集周期内，可能要消耗大量 CPU 资源。两种方法的区别如图 6-9 所示。

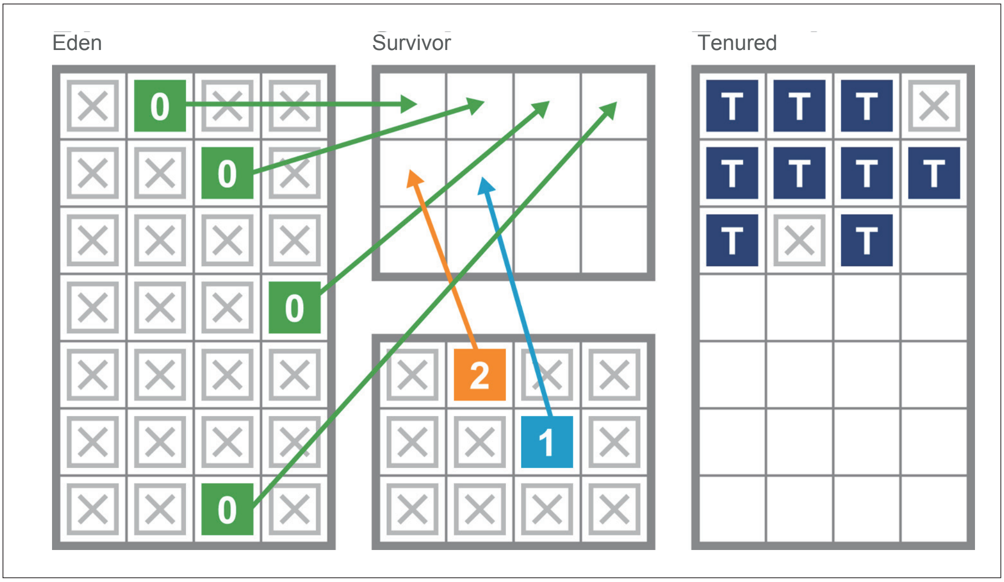


图 6-9：疏散与压缩的对比

这两种内存空间的行为截然不同，因为它们用于不同的目的。新生代收集的目的是处理寿命较短的对象，所以新生代空间的占用率会随着垃圾收集事件中的分配和清除而发生巨大变化。

相比之下，老年代不会发生明显的变化。偶尔会有大对象直接在老年代中创建，但除此之外，空间只会在垃圾收集时才会发生变化——要么是对象从新生代中晋升过来，要么是老年代收集（Old GC）或整堆收集时全面的重新扫描和重新安排。

6.5.3　并行收集器的局限性

并行收集器不仅会一次性处理整个代中的所有内容，而且会尽量提高收集效率。然而，这种设计也有一些缺点。首先，它们完全是全部停顿的。对于新生代而言，这通常不是问题，因为根据“弱分代假说”，只有很少的对象能够存活。



新生代并行收集器是这样设计的：死对象不会被触碰到，所以标记阶段的长度与存活对象的数量（少量）成正比。

这样的基本设计，再加上堆的新生代区域通常比较小，意味着对大部分工作负载而言，新生代收集的暂停时间会非常短。在一个内存为 2 GB 的现代 JVM 上（各部分采用默认大小），新生代收集典型的暂停时间可能只有几毫秒，而且经常少于 10 毫秒。

然而，老年代的收集大有不同。首先，老年代的大小默认为新生代的 7 倍。光是这一点，就会让一次整堆收集预期的全部停顿时间比新生代长得多。

另一个关键的事实是，标记时间与区域内的活对象的数量成正比。老年代中的对象可能会

很长寿，所以可能会有更多的老对象在一次整堆收集后仍然存在。

这种行为也解释了 ParallelOld 收集的一个关键弱点，即全部暂停时间将大致随着堆的大小呈线性扩展。随着堆的大小不断增加，ParallelOld 在暂停时间方面开始变得很糟糕。

有些刚接触垃圾收集理论的人，私下可能会有这样的想法，即对标记和清除算法进行微小的修改可能有助于缓解全部暂停时间，然而事实并非如此。40 多年来，计算机科学界对垃圾收集理论已经进行了充分的研究，目前还没有发现“你就不能……”这样的改进。

正如将在第 7 章中看到的，确实存在几款并发收集器，它们在运行的时候可以大大减少暂停时间。然而，它们并不是万能的，垃圾收集仍然存在一些基本的困难。

下面通过一个例子来看一下垃圾收集的原生实现方法存在的一个主要困难，让我们来考虑 TLAB 分配。TLAB 机制可以大大提升分配的性能，但是对收集周期并没有什么帮助。要了解原因，请考虑以下代码：

```
public static void main(String[] args) {
    int[] anInt = new int[1];
    anInt[0] = 42;
    Runnable r = () -> {
        anInt[0]++;
        System.out.println("Changed: "+ anInt[0]);
    };

    new Thread(r).start();
}
```

变量 `anInt` 是一个数组对象，其中只包含一个单个的 `int`。它从主线程持有的 TLAB 中分配，但紧接着又被传递给一个新线程。换句话说，TLAB 的关键属性，即它们对单个线程来说是私有的，只有在分配时才成立。这个属性基本上对象一旦分配完就会遭到破坏。

在 Java 环境中可以轻而易举地创建新线程，这是 Java 平台的基本能力，也是非常强大的能力。然而它使垃圾收集的情况变得更加复杂，因为有了新线程，就意味着有了新的执行栈，而其中的每一个栈帧都是 GC 根的来源。

6.6 分配的作用

Java 的垃圾收集过程通常在请求内存分配但手头没有足够的空闲内存来提供所需数量时触发。这意味着垃圾收集周期不是按照固定的或者可预测的时间表发生的，而是纯粹按照需求发生。

这也是垃圾收集最关键的一个方面：它不是确定性的，也不会以固定的节奏发生。相反，当堆中的一个或多个内存空间基本上已满，无法进一步创建对象时，就会触发一个垃圾收集周期。

> 这种按需进行的特性使得垃圾收集日志很难用传统的时间序列分析方法来处理。垃圾收集事件之间缺乏规律，让大部分时间序列库无法轻易适应。

当垃圾收集发生时，所有的应用程序线程都会被暂停（因为它们不能再创建任何对象，而且也不存在能运行很长时间而不用生成新对象的真实 Java 代码）。JVM 会接管所有的 CPU 核心来执行垃圾收集，并在重新启动应用程序线程之前回收内存。

为了更好地理解为什么分配如此重要，让我们考虑下面这个高度简化的案例研究。堆参数的设置情况如表 6-1 所示，假设它们不会随着时间的推移而改变。当然，一个真实的应用程序通常会动态调整堆的大小，但这个例子仅作简单说明之用。

表6-1：堆参数设置

堆 区 域	大 小
总体	2 GB
老年代	1.5 GB
新生代	500 MB
Eden	400 MB
S1	50 MB
S2	50 MB

在应用程序达到稳定状态之后，我们观察到以下垃圾收集指标：

分配率	100 MB/s
新生代垃圾收集时间	2 ms
整堆收集时间	100 ms
对象生命周期	200 ms

这表明，Eden 空间会在 4 秒内被填满，所以在稳定状态下，新生代收集每 4 秒就会发生一次。Eden 空间一填满，就会触发垃圾收集。Eden 空间中的大部分对象是死的，但是仍然活着的对象都会被疏散到一个 Survivor 空间中（为方便讨论，我们称其为 SS1）。在这个简单的模型中，因为对象的生命周期是 200 毫秒，所以在过去 200 毫秒内创建的任何对象，寿命都还没有结束，所以它们将存活下来。因此我们有：

GC0 @ 4 s 20 MB Eden → SS1 (20 MB)

又过了 4 秒，Eden 空间再次被填满，其对象需要继续疏散（这次是到 SS2）。不过在这个简化的模型中，通过 GC0 进入到 SS1 中的对象都已经不存在了，因为它们的生命周期只有 200 毫秒，而现在已经过去了 4 秒，所以所有在 GC0 之前分配的对象都已经死了。我们现在有：

GC1 @ 8.002 s 20 MB Eden → SS2 (20 MB)

换种说法，在 GC1 之后，SS2 中存在的都是刚从 Eden 区域过来的对象，SS2 中对象的年龄都不会大于 1。再继续进行一次收集，模式应该就很清楚了：

GC2 @ 12.004 s 20 MB Eden → SS1 (20 MB)

这种理想化的简单模型导致了这样一种情况，即没有任何对象能够晋升到老年代；在整个运行过程中，老年代一直是空着的。当然，这是非常不现实的。

相反，弱分代假说表明，对象的生命周期将是某种分布，而且由于分布的不确定性，有些对象最终会存活下来并进入老年代。

接下来看一个针对这种分配场景的一个非常简单的模拟器。它分配的对象，大部分寿命很短，但也有一些寿命相当长。它有一组参数：定义分配的 x 和 y，每个对象的大小都在这两个值之间；分配率（mbPerSec）；短命对象的寿命（shortLivedMS）；以及应用程序应该模拟的线程的数量（nThreads）。默认值见如下的代码清单：

```
public class ModelAllocator implements Runnable {
    private volatile boolean shutdown = false;

    private double chanceOfLongLived = 0.02;
    private int multiplierForLongLived = 20;
    private int x = 1024;
    private int y = 1024;
    private int mbPerSec = 50;
    private int shortLivedMs = 100;
    private int nThreads = 8;
    private Executor exec = Executors.newFixedThreadPool(nThreads);
```

省略了 main() 和其他任何启动 / 参数设置代码之后，ModelAllocator 的其余部分如下：

```
public void run() {
    final int mainSleep = (int) (1000.0 / mbPerSec);

    while (!shutdown) {
        for (int i = 0; i < mbPerSec; i++) {
            ModelObjectAllocation to =
                new ModelObjectAllocation(x, y, lifetime());
            exec.execute(to);
            try {
                Thread.sleep(mainSleep);
            } catch (InterruptedException ex) {
                shutdown = true;
            }
        }
    }
}

// 用来模拟弱分代假说的简单函数
// 返回一个对象的预期寿命——一般来说比较短暂，
// 不过也有“寿命较长”的一个小概率
    public int lifetime() {
        if (Math.random() < chanceOfLongLived) {
            return multiplierForLongLived * shortLivedMs;
        }

        return shortLivedMs;
    }
}
```

分配器主运行器（runner）与一个简单的模拟对象（mock）结合，该对象用来表示应用程序执行的对象分配：

```
public class ModelObjectAllocation implements Runnable {
    private final int[][] allocated;
    private final int lifeTime;

    public ModelObjectAllocation(final int x, final int y, final int liveFor) {
        allocated = new int[x][y];
        lifeTime = liveFor;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(lifeTime);
            System.err.println(System.currentTimeMillis() +": "+ allocated.length);
        } catch (InterruptedException ex) {
        }
    }
}
```

用 VisualVM 查看时，它将显示简单的锯齿模式，我们经常在能高效利用堆的 Java 应用程序的内存行为中观察到这种模式，如图 6-10 所示。

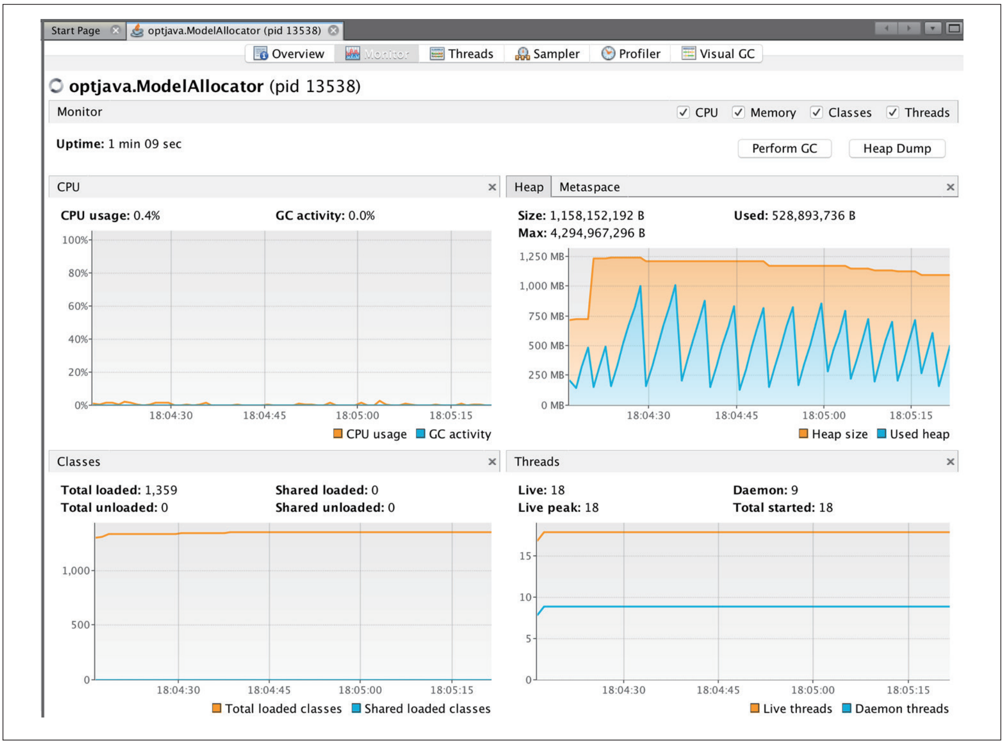


图 6-10：简单的锯齿模式

第 7 章将对内存情况的工具化和可视化进行更多的讨论。感兴趣的读者可以在讨论调优之前先行下载本章提及的分配和生命周期模拟器，并设置参数以查看分配率和长寿对象百分

比的作用。

接下来将介绍分配行为中一个非常常见的方面，并以此结束对分配的讨论。在现实世界中，分配率可以是高度可变的和“突发性”的。对具有如上所述的稳定状态行为的 Java 应用程序，请考虑以下情形：

2 s	Steady-state allocation	100 MB/s
1 s	Burst/spike allocation	1 GB/s
100 s	Back to steady-state	100 MB/s

最初的稳定状态的执行在 Eden 空间中分配了 200 MB。没有长寿对象，所以这类内存的生命周期都是 100 毫秒。接下来，分配峰值出现了。分配 Eden 空间中另外的 200 MB 只需要 0.2 秒，其中 100 MB 的对象还没有达到 100 毫秒的年龄阈值。存活对象所需的空间要大于 Survivor 空间，所以 JVM 别无选择，只能让这些对象直接晋升到老年代。所以我们得到：

GC0 @ 2.2 s 100 MB Eden → Tenured (100 MB)

分配率的急剧增加产生了 100 MB 的存活对象。不过要注意的是，在这个模型中，所有的“存活对象”实际上寿命都不长，而且很快就会变成死对象，它们会让老年代变得比较杂乱。直到下一次发生整堆收集时它们才会被回收。

再继续进行几次收集，模式就清晰了：

GC1 @ 2.602 s   200 MB Eden → Tenured (300 MB)
GC2 @ 3.004 s   200 MB Eden → Tenured (500 MB)
GC2 @ 7.006 s   20 MB Eden → SS1 (20 MB) [+ Tenured (500 MB)]

请注意，如前所述，垃圾收集器根据需要运行，而不是定期运行。分配率越大，垃圾收集就越频繁。如果分配率过高，那么对象最终会被强制提前晋升。

这种现象被称为**过早晋升**（premature promotion）；它是垃圾收集器最重要的间接效应之一，也是许多调优练习的一个起点，我们将在下一章介绍。

6.7　小结

自平台诞生以来，垃圾收集一直是 Java 社区内讨论的热门话题。本章介绍了性能工程师需要了解的关键概念，以便有效地与 JVM 的垃圾收集子系统一起工作。这些概念包括：

- 标记和清除收集；
- 对象在 HotSpot 内部的运行时表示；
- 弱分代假说；
- HotSpot 的内存子系统实例；
- 并行收集器；
- 分配及其核心作用。

下一章将讨论垃圾收集的调优、监控和分析。有些主题在本章已经出现过，特别是分配，以及过早晋升等特殊效应，这些内容对于接下来的目标和主题也特别重要，经常回来参考本章可能会很有帮助。

第7章

垃圾收集高级话题

上一章介绍了 Java 垃圾收集的基本理论。以此为起点，本章将进一步研究现代 Java 垃圾收集器的理论。这个领域有很多不可避免的权衡可以指导工程师如何选择收集器。

首先，本章将介绍并深入了解 HotSpot JVM 提供的其他收集器，其中包括停顿时间超短、通常为并发的收集器（CMS）和现代通用收集器（G1）。

此外，还将考虑一些比较少见的收集器，包括：

- Shenandoah
- C4
- 均衡（balanced）收集器
- 遗留的 HotSpot 收集器

并非所有这些收集器都在 HotSpot 虚拟机中使用，我们还将讨论另外两个虚拟机的收集器：IBM J9（IBM 的一款 JVM，之前是闭源的，目前正在逐步开源）和 Azul Zing（一款专有的 JVM），我们在 2.6 节曾介绍过。

7.1 权衡与可插拔的收集器

就 Java 平台而言，有一点可能初学者未必能马上意识到，即尽管 Java 有垃圾收集器，但是语言和虚拟机规范都没有说明应该如何实现垃圾收集。事实上，曾经有一些 Java 实现（例如 Lego Mindstorms）根本没有实现任何类型的垃圾收集！[1]

注 1：在这样的系统中编程是非常困难的，因为创建的每一个对象都必须复用，任何超出作用域的对象事实上都会导致内存泄漏。

在 Sun（现在是 Oracle）的环境中，垃圾收集子系统被视为一个可插拔的子系统。这意味着同一个 Java 程序可以使用不同的垃圾收集器来执行，而不改变程序的语义，尽管它的性能可能会根据使用的收集器而有很大的不同。

使用可插拔的收集器的主要原因是垃圾收集是一种通用的计算技术。特别是，同样的算法未必适用于每个工作负载。由于人们关注点不同，有些还是相互冲突的，因此垃圾收集算法代表了一种妥协或权衡。

没有一种通用的垃圾收集算法可以同时针对所有垃圾收集问题进行优化。

在选择垃圾收集器时，开发人员通常需要考虑以下主要问题：

- 暂停时间（也叫暂停长度或持续时间）；
- 吞吐量（垃圾收集时间占应用程序运行时间的百分比）；
- 暂停频率（收集器需要停止应用程序的频率）；
- 回收效率（一个垃圾收集工作周期内可收集的垃圾量）；
- 暂停的一致性（所有的暂停长度是否大致相同）。

其中，暂停时间往往会引起过多的关注。尽管暂停时间对许多应用程序确实很重要，但也不应该孤立地考虑它。

对许多工作负载来说，暂停时间并不是一个有效或有用的性能特性。

例如，高度并行的批处理或大数据应用程序可能更关注吞吐量，而不是暂停时间的长短。对于很多批处理作业来说，暂停时间就算有几十秒也没有多大关系。因此，与不计成本地降低暂停时间的算法相比，应该首选 CPU 效率和吞吐量更高的垃圾收集算法。

性能工程师还需要注意的是，在考虑收集器的选择时，有时还会有一些其他的权衡和顾虑。在 HotSpot 的情况下，选择还受制于可用的收集器。

截至 Java 10，在 Oracle/OpenJDK 内有 3 种主流的收集器供一般性生产环境使用。我们已经见过并行（又称吞吐量）收集器，从理论和算法的角度看它们是最容易理解的。本章会介绍另外两种主流的收集器，并解释它们与并行垃圾收集的区别。

在本章最后，从 7.5 节开始到后面几节，我们还将认识一些其他可用的收集器，但请注意，并不是所有这些收集器都推荐用于生产环境，有些现在已经废弃了。本章还将简单讨论在非 HotSpot JVM 中可用的收集器。

7.2 并发垃圾收集理论

在专门的系统中，例如图形或动画显示系统，通常有一个固定的帧率，这为执行垃圾收集提供了一个有规律的、固定的机会。

然而，通用的垃圾收集器并没有这样的领域知识来提高暂停时间的确定性。更为糟糕的是，不确定性是由分配行为直接导致的，而很多使用 Java 的系统显示出高度可变的分配行为。

> 这种安排的次要缺点是本身的计算被延迟了；其主要缺点是这些垃圾收集穿插而来的不确定性。[2]
>
> ——Edsger Dijkstra

现代垃圾收集理论的出发点就是试图解决 Dijkstra 所提出的问题，即 STW 暂停的不确定性（无论是持续时间还是频率）是使用垃圾收集技术的主要烦恼。

一种解决方法是使用一个并发（至少部分并发，或大部分并发）的收集器，在应用程序线程运行的同时做一些收集所需的工作，以减少暂停时间。这不可避免地降低了用于应用程序实际工作的计算能力，同时也使执行收集所需的代码复杂化了。

不过在讨论并发收集器之前，还需要介绍一个重要的垃圾收集术语和技术，因为它对理解现代垃圾收集器的性质和行为至关重要。

7.2.1 JVM安全点

为了执行 STW 垃圾收集，比如 HotSpot 的并行收集器所执行的那些，必须停止所有的应用程序线程。看上去我们在反复说同样的事情，但直到现在我们还没有讨论 JVM 到底是**如何**实现这一点的。

> 实际上，JVM 并不是一个完全抢占式的多线程环境。
>
> ——一个秘密

这并不意味着它是一个纯粹的合作式环境，恰恰相反，操作系统仍然可以在任何时候抢占线程（从某个 CPU 核心上移除一个线程）。例如当一个线程用尽了其时间片，或将自己放入 `wait()` 中时，就会出现这种情况。

除了这个核心的操作系统功能外，JVM 还需要执行协调操作。为了实现这一点，运行时就要求每个应用程序线程都有称为**安全点**（safepoint）的特殊执行点，其中线程的内部数据结构处于已知的良好状态。此时线程能被挂起，以执行协调操作。

> 在 STW 垃圾收集（经典示例）和线程同步中可以看到安全点的作用，不过在其他场景中也可以看到。

注 2：这句话出自 7.2.2 节提到的有关三色标记的论文。——译者注

要理解安全点的意义，可以考虑一个完全 STW 垃圾收集器的情况。为了使之运行，它需要一个稳定的对象图。这意味着所有的应用程序线程都必须暂停，因为垃圾收集线程无法要求操作系统对应用程序线程强制执行这个要求，所以必须有应用程序线程（它作为 JVM 进程的一部分执行）的配合才能实现。JVM 对安全点的实现方式受限于以下两个主要规则：

- JVM 不能强制一个线程进入安全点状态；
- JVM 可以阻止一个线程离开安全点状态。

这意味着当需要安全点时，JVM 解释器的实现必须包含在屏障（barrier）处释放（yield）的代码。对于 JIT 编译的方法，必须在生成的机器码中插入等效的屏障。到达安全点的一般情况是以下这样的。

1. JVM 设置一个全局的“安全点时间”（time to safepoint）标志。
2. 单个的应用程序线程轮询并查看该标志已设置。
3. 暂停并等待被再次唤醒。

当这个标志被设置后，所有的应用程序线程都必须停止，而且停得快的线程必须等待停得慢的，这个时间可能不会被完全统计到暂停时间中。

普通的应用程序线程使用这种轮询机制：在解释器中，它们总是会在执行任何两个字节码之间进行检查。如果是编译后的代码，最常见的一种情况是 JIT 编译器在退出编译方法的地方插入一个安全点轮询，另一种情况是循环分支向后跳转时（比如，回到循环的顶部）插入一个安全点轮询。

一个线程可能要花很长时间才到安全点，甚至理论上可能永远不会停止，但这是一种病态的情况，一定是有意而为之的。

> 所有的线程都必须在 STW 阶段开始之前完全停止，这个想法类似于使用锁存器，比如用 `java.utilite.concurrent` 库中的 `CountDownLatch` 实现的锁存器。

这里值得一提的是安全点条件的一些特殊情况。在如下情况下，一个线程会自动处于安全点状态：

- 阻塞在一个管程（monitor）上；
- 正在执行 JNI 代码。

在如下情况下，一个线程**未必**处于安全点状态：

- 在执行字节码的过程中（解释模式）；
- 已经被操作系统中断。

稍后还会再讲到安全点机制，因为它是 JVM 内部运作的关键一环。

7.2.2　三色标记

Dijkstra 和 Lamport 在 1978 年发表的论文中描述了**三色标记**算法，该论文对证明并发算法和垃圾收集的正确性具有里程碑式的意义，其中所描述的基本算法至今仍是垃圾收集理论

的重要组成部分。[3]

三色标记算法的工作原理如下：

- 垃圾收集根标记为灰色；
- 所有其他对象都标记为白色；
- 有一个标记线程移动到一个随机的灰色节点；
- 如果该节点有任何白色的子节点，那标记线程首先会将这样的子节点标记为灰色，然后将该节点标记为黑色；
- 这个过程重复进行直到再也没有灰色节点；
- 所有标记为黑色的对象，都已被证明是可达的，并且必须保持存活状态；
- 白色节点符合收集条件，对应不再可达的对象。

虽然有些复杂，但这就是算法的基本形式，示例如图 7-1 所示。

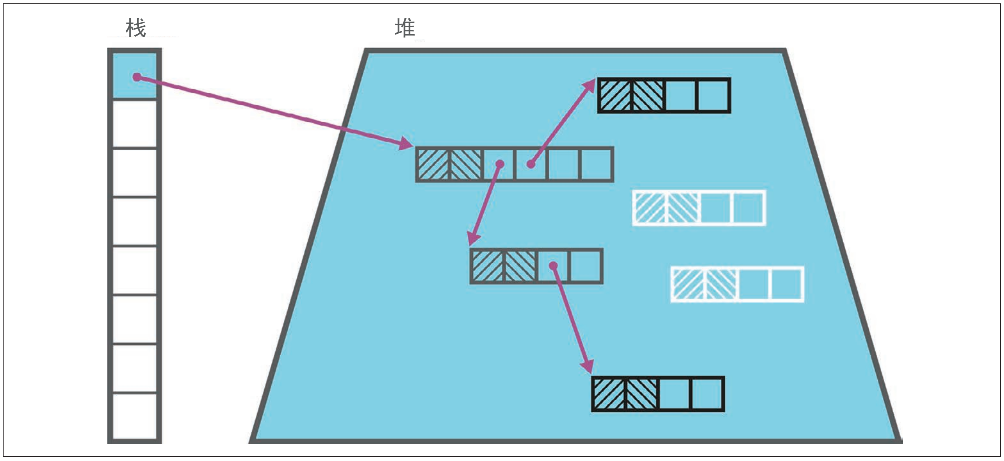


图 7-1：三色标记

并发收集经常使用一种叫作**原始快照**（snapshot at the beginning，SATB）的技术。这意味着如果对象在收集周期开始时是可达的，或者那时已经分配，则收集器会将其视为活的。这就给算法增加了一些小麻烦。比如，当 Mutator[4] 线程需要创建新对象时，如果此时垃圾收集正在运行，那么这些对象需要标记为黑色，如果没有，则它们需要标记为白色。

三色标记算法还需要多做一些工作，以确保由正在运行的应用程序线程引入的修改不会导致活对象被收集。这是因为在并发收集器中，标记线程在执行三色算法的同时，应用（Mutator）线程正在改变对象图。

考虑这样的情况：一个对象已经被标记线程标记为黑色，然后又被 Mutator 更新，指向了一个白色的对象，如图 7-2 所示。

注 3：Edsger Dijkstra，Leslie Lamport，A. J. Martin, C. S. Scholten and E. F. M. Steffens，“On-the-Fly Garbage Collection: An Exercise in Cooperation,” *Communications of the ACM* 21（1978），966–975。

注 4：Mutator 线程和 Collector 线程相对，一般指的是会修改内存状态的用户线程。——译者注

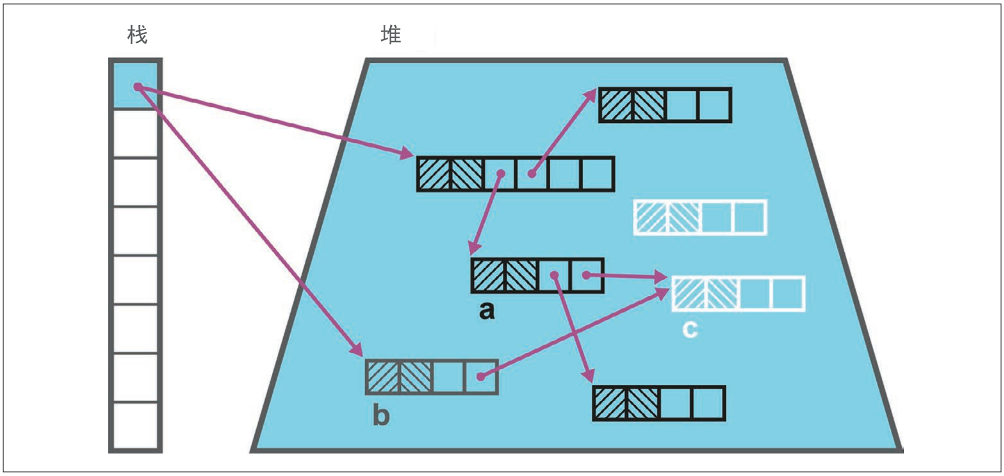


图 7-2：Mutator 线程可能导致三色标记失效

如果从灰色对象指向新的白色对象的引用现在都被删除了，那就会出现这样一种情况，即这个白色对象应该仍然是可达的，但是它会被删除，因为根据算法的规则，这个对象不会被找到。

这个问题可以通过几种不同的方式来解决。比如我们可以将黑色对象的颜色改回灰色，在 Mutator 线程处理更新时，将其加入到需要处理的节点集合中。

该方法使用了一个“写屏障”来进行更新，而且它还有一个很好的算法特性，那就是在整个标记周期中能保持**三色不变性**（tri-color invariant）。

> 在并发标记期间，任何黑色对象节点都不能持有指向白色对象节点的引用。
>
> ——三色不变性

另一种方法是使用一个队列，保存所有可能会破坏三色不变性的修改，然后在主阶段结束后运行一个“修正”阶段。不同的收集器可以根据性能或者所需的加锁的数量等标准，用不同方式来解决三色标记的这个问题。

下一节将介绍低延迟的收集器——CMS。虽然 CMS 的适用范围有限，但我们会在其他收集器之前先介绍它。这是因为开发人员往往意识不到，垃圾收集调优需要在多大程度上进行权衡和妥协。

通过先考虑 CMS，可以揭示性能工程师在考虑垃圾收集时应该注意的一些实际问题。希望这样做能让我们更多的是基于证据进行调优，基于固有的一些权衡来选择收集器，而不是按照坊间传说调优。

7.3 CMS

CMS 收集器是专为老年代空间设计的一个延迟极低的收集器，它通常会与一个稍微修改过的、用于 Young GC 的并行收集器（叫作 ParNew，而不是 Parallel GC）配对使用。

CMS 会在应用线程仍在运行的时候尽量多做一些工作，以便最大化地减少暂停时间。它使用的标记算法是三色标记，当然这就意味着在收集器正在扫描堆的同时，对象图可能会被修改。因此，CMS 必须对其记录进行修正，以避免破坏垃圾收集器的第二条规则，也就是把仍然活着的对象收集了。

这就导致了与并行收集器相比，CMS 所需的阶段更为复杂。这些阶段如下。

1. 初始标记（initial mark）（STW）
2. 并发标记（concurrent mark）
3. 并发预清理（concurrent preclean）
4. 重新标记（remark）（STW）
5. 并发清除（concurrent sweep）
6. 并发重置（concurrent reset）

垃圾收集在大多数阶段是与应用程序线程同时运行的，但是在初始标记和重新标记这两个阶段中，所有应用程序线程都必须停止。总的效果应该是用两个通常来说时间比较短的 STW 暂停来代替一次长时间的 STW 暂停。

初始标记阶段的目的是为该区域内的垃圾收集提供一个稳定的起点集合，这些起点被称为**内部指针**，相当于用于收集周期的垃圾收集根。这种方法的优点是它使得标记阶段可以专注于单个垃圾收集池，而不必考虑其他内存区域。

初始标记阶段结束后，并发标记阶段开始。本质上就是在堆上运行三色标记算法，记录以后可能需要修正的任何修改。

并发预清理阶段似乎试图尽可能地缩短会造成 STW 的重新标记阶段的长度。重新标记阶段使用卡表来修正可能会在并发标记阶段受 Mutator 线程影响的标记。

对大多数工作负载而言，使用 CMS 可以观测到如下影响：

- 应用程序线程不会停顿很久；
- 一次 Full GC 周期需要更多时间（以挂钟时间计算）；
- 当 CMS 的垃圾收集周期运行的时候，应用程序的吞吐量会降低；
- 垃圾收集会使用更多的内存来记录对象信息；
- 整体来看，执行垃圾收集需要更多的 CPU 时间；
- CMS 不会对堆进行压缩，所以老年代的碎片会越来越多。

细心的读者会注意到，并不是所有这些影响都是正面的。请记住，垃圾收集并没有什么良方，只有针对工程师正在调优的具体工作负载所做出的一系列适当（或可以接受）的选择。

7.3.1　CMS是如何工作的

令人诧异的是，CMS 最常被忽视的一个方面是它的巨大优势，它在大多数时候是与应用程序线程并发运行的。CMS 会默认使用一半的可用线程来执行垃圾收集的并发阶段，并将另一半留给应用线程来执行 Java 代码，**而这就不可避免地会涉及分配新对象**。听上去很简单，但它有一个直接后果，即如果 Eden 区在 CMS 运行时被填满了，那会发生什么？

答案并不令人吃惊，因为应用程序线程无法继续，所以它们会暂停，并且在 CMS 运行时触发一次会导致 STW 的新生代收集（Young GC）。这次 Young GC 运行的时间通常会比并行收集器长，因为只有一半的 CPU 核心可用于 Young GC，另一半还在运行 CMS。

在 Young GC 结束时，通常有些对象会有资格晋升到 Tenured 区。因为这些对象需要在 CMS 仍在运行的时候移动到 Tenured 区，所以两个收集器之间需要做一些协调工作。这就是 CMS 需要一个略有不同的新生代收集器的原因。

正常情况下，Young GC 只会把很少量的对象晋升到 Tenured 区，而 CMS 老年代收集也能正常完成，释放 Tenured 区的空间。之后应用程序回归正常处理，所有的 CPU 核心也都可以释放出来供应用程序线程使用了。

但是，在分配率非常高时，可能在 Young GC 中导致过早晋升（在 6.6 节的最后讨论过）。这会引发一种情况，即 Young GC 有太多对象要晋升到 Tenured 区的可用空间中，如图 7-3 所示。

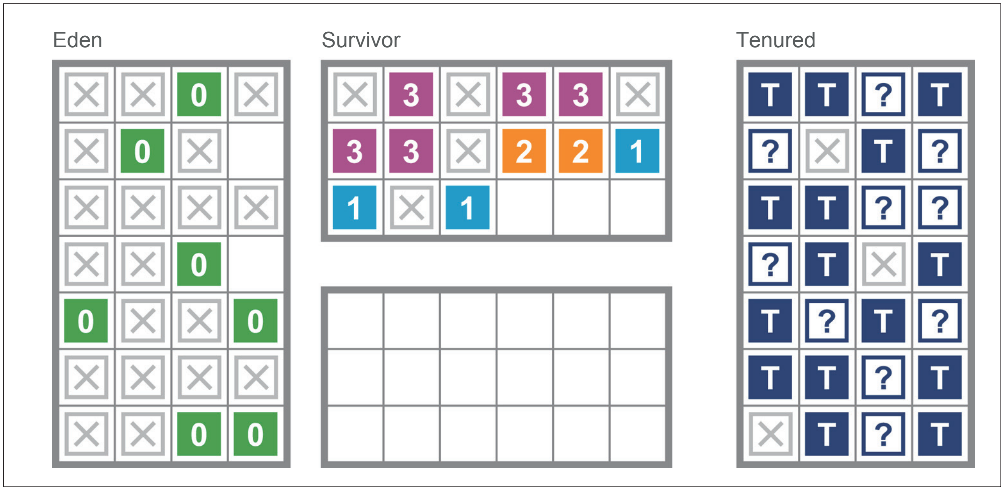


图 7-3：分配压力下的并发模式失败

这种情况被称为**并发模式失败**（concurrent mode failure，CMF），除了回退到使用会造成 STW 的 ParallelOld，此时 JVM 别无选择。实际上，分配压力如此之大，以致在所有的“净空间”被新晋升的对象填满之前，CMS 没有时间完成老年代的处理。

为了避免频繁出现并发模式失败，CMS 需要在 Tenured 区完全填满之前启动一次收集。至于在 Tenured 堆的占用达到何种水平时 CMS 将会启动，可以通过观察堆的行为来控制。它可能会受到开关的影响，初始值默认为 Tenured 区的 75%。

堆的碎片化也会导致并发模式失败。与 ParallelOld 不同，CMS 在运行过程中不会对 Tenured 区进行压缩。这意味着在 CMS 运行完成后，Tenured 区中的空闲空间并不是一个连续的块，晋升而来的对象必须被填充到现有对象之间的空隙中。

在某一时刻，Young GC 可能会遇到这样一种情况：由于 Tenured 区缺乏足够的连续空间来

复制对象，因此对象无法晋升过去，如图 7-4 所示。

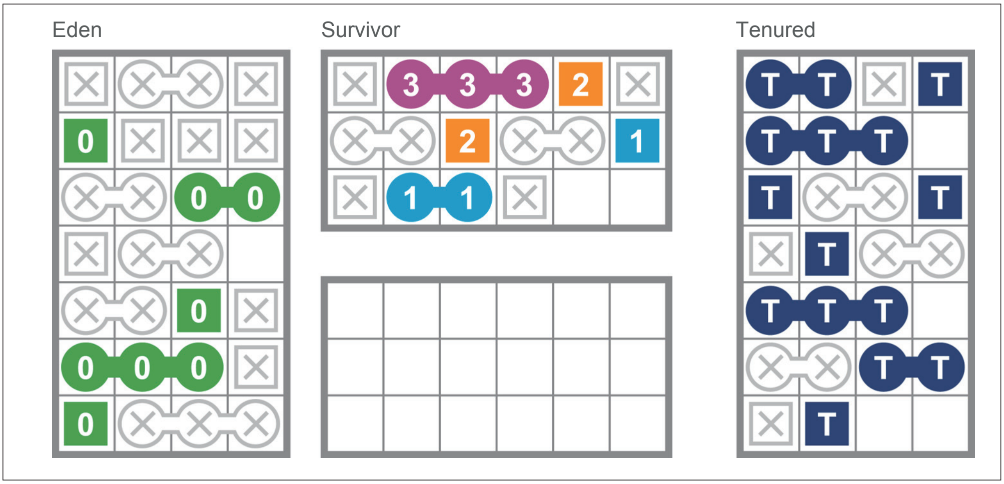


图 7-4：碎片化导致的并发模式失败

这就是由堆的碎片化引起的并发模式失败，和之前一样，唯一的解决办法是回退到使用 ParallelOld（这是压缩的）进行一次 Full GC，从而释放出足够的连续空间以支持对象晋升。

无论是堆碎片化，还是 Young GC 的速度超过 CMS，需要回退到完全 STW 的 ParallelOld 收集对应用程序而言都是重大事件。事实上，为避免遭受 CMF 而对使用 CMS 的低延迟应用程序进行调优本身就是一个重要的课题。

在内部，CMS 使用了一个内存**块**的空闲列表来管理可用内存。在最后的并发清除阶段，连续的空闲块将被 sweeper 线程合并。这是为了提供更大的空闲空间块，以避免由碎片化造成的 CMF。

然而，sweeper 会与 mutator 并发运行。因此，除非 sweeper 和分配器线程正确同步，否则刚分配的块可能会被错误地清除掉。为了防止出现这种情况，sweeper 在清扫过程中会锁住空闲列表。

7.3.2 用于CMS的基本JVM标志

CMS 收集器可用以下标志开启：

```
-XX:+UseConcMarkSweepGC
```

在现代版本的 HotSpot 上，这个标志也会激活 `ParNewGC`（与并行新生代收集器略有不同的变种）。

总的来说，CMS 提供了大量可以调整的标志（超过 60 个）。有时进行基准测试很有吸引力，因为它试图通过仔细调整 CMS 提供的不同选项来优化性能。千万要抵制这种诱惑，因为在大部分情况下，这实际上是“忽略大局”或“按照坊间传说调优”这些反模式的伪装（参见 4.4 节）。

我们在 8.4 节会详细介绍 CMS 的调优。

7.4 G1

G1 是一款与并行收集器或 CMS 的风格都非常不同的收集器。它最初以高度实验性和不稳定的状态出现于 Java 6 中，但在 Java 7 的整个开发过程中经过了大量重写，直到 Java 8u40 的发布才真正成为稳定的、可供生产使用的版本。

> 无论考虑什么类型的工作负载，如果想使用 G1，不建议采用任何 Java 8u40 之前的版本。

G1 最初是打算成为一款替代用的低延迟收集器，具有如下特性。

- 调优比 CMS 更容易；
- 不容易受到过早晋升的影响；
- 行为在大堆上能够更好地扩展（特别是暂停时间）；
- 能够消除（或者能极大减少回退到）完全 STW 的收集。

然而，随着时间的推移，G1 逐渐被认为是一款通用的收集器，在更大的堆上有更少的暂停时间（越来越多的人认为这是“新常态”）。

> 不管对最终用户的影响如何，Oracle 都坚持让 G1 取代并行收集器，成为 Java 9 中的默认收集器。因此，性能分析人员需要充分理解 G1，并且任何要从 Java 8 迁移到 Java 9 的应用程序都需要恰当地进行重新测试，这应该是迁移工作的一部分。

G1 收集器在设计上重新思考了到目前为止我们都在用的分代的概念。与并行或 CMS 收集器不同，G1 中的“代”没有专用的连续内存空间。此外，它也没有使用半空间堆布局。

7.4.1 G1堆布局和区域

G1 堆基于区域（region）的概念。这些区域的大小默认为 1 MB（在更大的堆上会更大）。区域的使用支持非连续的分代，从而使收集器可以不需要在每次运行时收集所有垃圾。

> 整体的 G1 堆在内存中仍然是连续的，只是组成每一代的内存不必再是连续的。

G1 堆基于区域的布局如图 7-5 所示。

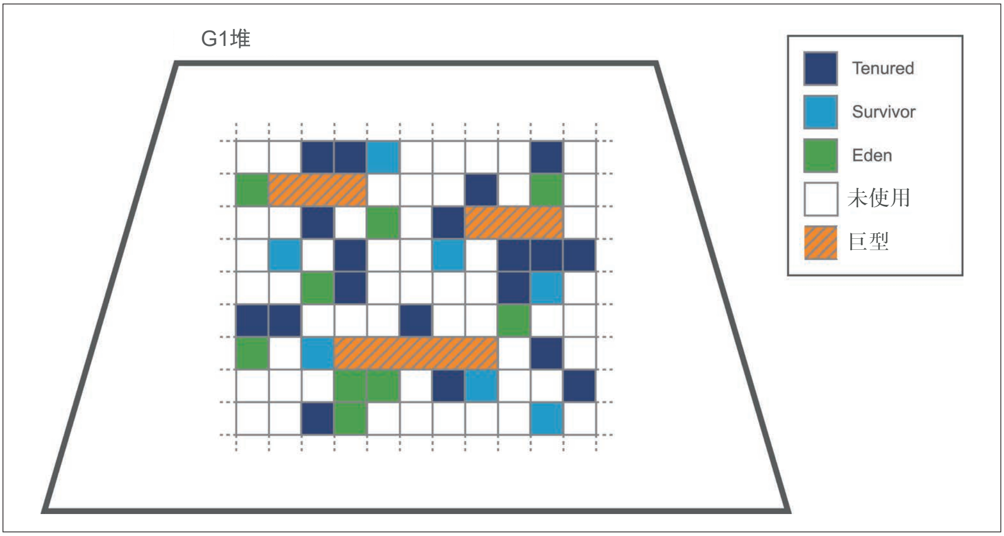


图 7-5：G1 的区域

G1 的算法支持区域的大小为 1、2、4、8、16、32 或 64 MB 中的某个值。默认情况下，它期望堆中区域的数量在 2048 到 4095 之间，如果不在，它会调整区域的大小来实现这个目标。

要计算区域大小，可以计算 <Heap size> / 2048 这个值，并将结果取整数，选择离它最近的所允许的区域大小。区域的数量可以这样计算：

```
Number of regions = <Heap size> / <region size>
```

照例，可以通过运行时开关来修改这个值。

7.4.2　G1算法设计

从上层看这款收集器，可以得到以下信息：

- G1 使用了一个并发标记阶段；
- G1 是一款疏散收集器；
- G1 提供了“统计型压缩”（statistical compaction）。

在预热的同时，收集器会跟踪并统计每个垃圾收集执行周期有多少“典型”的区域可被收集。如果能收集足够的内存以平衡自上次垃圾收集以来分配的新对象，那 G1 就不会因为分配而失败。

TLAB 分配、疏散到 Survivor 空间以及晋升到老年代区，这些概念与我们已经接触过的其他 HotSpot 垃圾收集大体相似。

> 占用空间超过区域一半的对象被认为是巨型（humongous）对象。它们被直接分配在特殊的**巨型区域**中，该区域是空闲的连续区域，可以立即成为 Tenured 区（而非 Eden 区）的一部分。

G1 仍然有由 Eden 和 Survivor 区域组成的新生代这个概念，当然，在 G1 中组成的代的区域是不连续的。新生代的大小是自适应的，会基于整体的暂停时间目标来调整。

回想一下，在介绍 ParallelOld 收集器时，我们曾在 6.3 节讨论过“有极少数从老年代指向新生代对象的引用”。HotSpot 使用一种叫作卡表的机制来帮助在并行和 CMS 收集器中利用这种现象。

G1 收集器有一个相关的功能来帮助区域进行跟踪。**记忆集**（remembered set，通常只是称为 RSet）是每个区域都有的条目，它们会记录指向当前堆区域的外部引用。这意味着 G1 不需要通过遍历整个堆来寻找指向某个区域的引用，而只需要检查 RSets，然后扫描这些区域来寻找引用。

图 7-6 演示了 G1 是如何使用 RSets 来实现分配器和收集器之间的工作划分的。

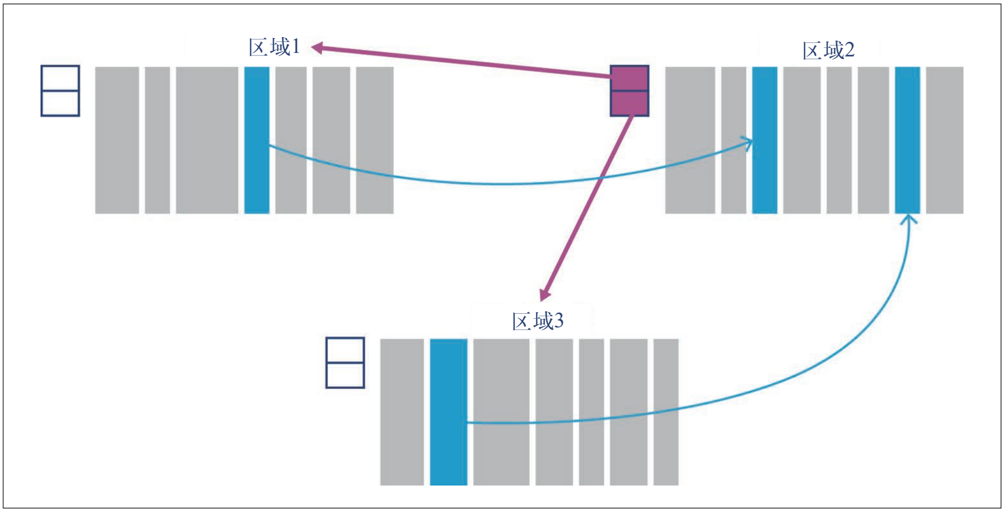


图 7-6：记忆集

RSets 和卡表这两种方法都可以帮助处理一个叫**浮动垃圾**（floating garbage）的垃圾收集问题。浮动垃圾产生的原因是有些对象本来应该是死的，但是在当前收集集合之外的死亡对象中仍然保留着对这些对象的引用。也就是说，如果是全局标记，那么可以看到这些对象是死的，但如果是一个局限性更大的局部标记，那可能会错误地报告这些对象是活的，这和当前使用的根集有关。

7.4.3　G1的各阶段

G1 可以划分为一系列阶段，和之前遇到的收集器（特别是 CMS）有点类似，具体如下。

1. 初始标记（initial mark），STW
2. 并发根扫描（concurrent root scan）
3. 并发标记（concurrent mark）
4. 重新标记（remark），STW

5. 清理（cleanup），STW

并发根扫描是一个并发标记阶段，该阶段会扫描初始标记的 Survivor 区域以寻找指向老年代的引用。这个阶段必须在下一次 Young GC 开始之前完成。在重新标记阶段，标记周期完成。这个阶段还执行引用处理（包括弱引用和软引用），并进行与实现 SATB 方法有关的清理工作。

清理阶段大多是 STW 的，它包括处理记账信息和“擦洗”RSet。记账处理任务会识别现在已经完全空闲并准备好复用的区域（比如用作 Eden 区域）。

7.4.4 用于G1的基本JVM标志

在 Java 8 和更早的版本中，需要使用如下开关来启用 G1：

```
+XX:UseG1GC
```

回想一下，G1 是围绕**暂停时间目标**设计的。这使得开发人员可以指定应用程序在每个垃圾收集周期中应该暂停的最大时间。虽说是一个目标，但是 JVM 并不能保证应用程序能够达到这个目标。如果这个值设置得太低，那么垃圾收集子系统将无法达到目标。

> 垃圾收集由分配驱动，这对于许多 Java 应用程序来说是非常难以预测的，因为可能会限制或损害 G1 达到暂停时间目标的能力。

控制 G1 收集器这一核心行为的开关是：

```
-XX:MaxGCPauseMillis=200
```

这意味着默认的暂停时间目标是 200 毫秒。因此，在实践中很难可靠地实现暂停时间小于 100 毫秒，收集器也很难达到这样的目标。另一个可能有用的选择是修改区域的大小，可以这样覆盖默认算法：

```
-XX:G1HeapRegionSize=<n>
```

注意，`<n>` 必须是 2 的幂，范围在 1 到 64 之间，表示一个单位为 MB 的值。我们在第 8 章讨论 G1 调优时会介绍其他 G1 标志。

总的来说，G1 现在已经是一种稳定的算法了，并且得到了 Oracle 的完全支持（推荐从 8u40 开始）。对于真正的低延迟工作负载，大部分情况下其表现不如 CMS，也不清楚单纯在暂停时间这方面，G1 是不是已经能够挑战像 CMS 这样的收集器。不过，G1 收集器仍然在不断改进，它也是 Oracle 的 JVM 团队在垃圾收集上的重点工程方向。

7.5 Shenandoah

除了由 Oracle 主导的在下一代通用收集器方面的努力，Red Hat 也在 OpenJDK 项目中开发了自己的收集器 Shenandoah。这仍然是一款实验性的收集器，在编写本书时还没有准备好投入到生产中使用。不过，它显示了一些很有前景的特性，值得介绍一下。

和 G1 一样，Shenandoah 的目的也是减少暂停时间（尤其是在大堆上），对此的解决方法是执行并发压缩。在 Shenandoah 中收集的相应阶段如下。

1. 初始标记（initial mark），STW
2. 并发标记（concurrent marking）
3. 最终标记（final marking），STW
4. 并发压缩（concurrent compaction）

这些阶段最初看起来可能和在 CMS 与 G1 中看到的很像，因为 Shenandoah 使用了一些类似的方法（比如 SATB）。不过它们之间也有一些根本的区别。

Shenandoah 最显著和最重要的一个方面是它使用了 **Brooks 指针**。[5] 该技术使用每个对象的一个额外内存字来指示该对象在垃圾收集的上一个阶段是否被重新安置（relocate）了，并给出对象内容的新版本的位置。

Shenandoah 为其 oops 所使用的相应的堆布局如图 7-7 所示。这种机制有时被称为“转发指针”（forwarding pointer）方法。如果对象的位置没有调整，那么 Brooks 指针只是指向下一个内存字[6]。

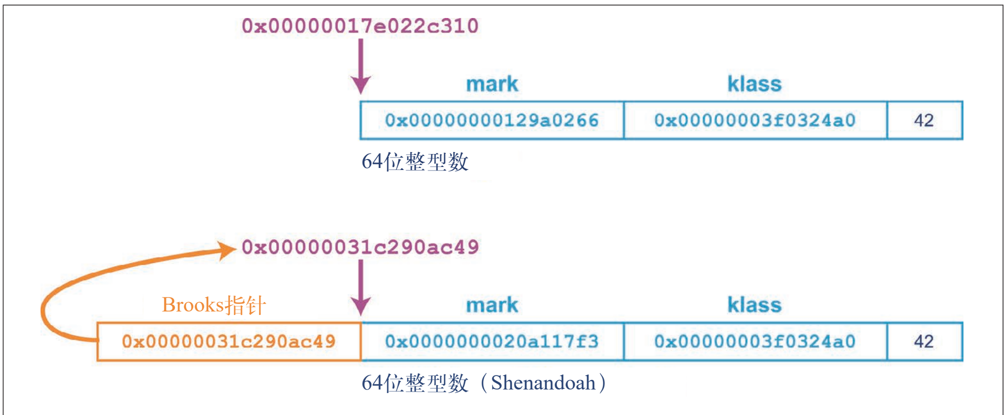


图 7-7：Brooks 指针

> Brooks 指针机制依赖于硬件比较和交换（compare-and-swap，CAS）操作的可用性来提供转发地址的原子更新。

并发标记阶段会跟踪整个堆，并标记任何活对象。如果某个对象引用指向了一个包含转发指针的 oop，则更新该引用，使其直接指向新的 oop 位置，如图 7-8 所示。

注 5：Rodney Brooks，“Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware，” in LFP’84，*Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*（New York: ACM, 1984），256–262。

注 6：即指向当前对象。——译者注

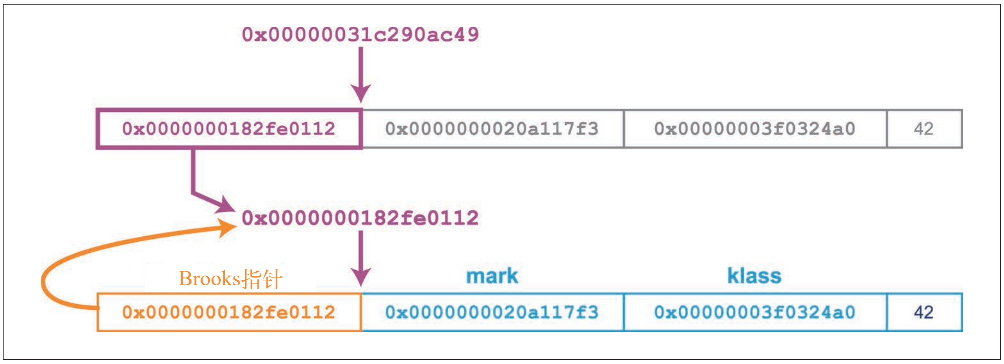


图 7-8：更新转发指针

在最终标记阶段，Shenandoah 会让应用程序 STW 以重新扫描根集，然后复制并更新根集使其指向疏散后的副本。

7.5.1 并发压缩

垃圾收集线程（它与应用程序线程并发执行）现在按照下列步骤执行疏散操作。

1. 将对象复制到某个 STAB（根据推测）。
2. 使用一个 CAS 操作来更新 Brooks 指针，使其指向推测的副本。
3. 如果成功，则压缩线程赢得比赛，那么以后都将通过 Brooks 指针对这个版本的对象进行所有访问。
4. 如果失败，则压缩线程就输得了比赛，那么它会撤销推测性副本，然后使用获胜线程留下的 Brooks 指针。

由于 Shenandoah 是一款并发收集器，因此当收集周期运行时，应用程序线程会产生更多的垃圾。所以，在应用程序运行期间，收集应与分配同步进行。

7.5.2 获取Shenandoah

Shenandoah 收集器目前还没有成为 Oracle Java 构建版本的一部分，也没有在大多数 OpenJDK 发行版中提供。不过在一些 Linux 发行版（包括 Red Hat Fedora）中，它已经作为 IcedTea 二进制文件的一部分提供了。

在撰写本书时，其他用户还必须从源代码编译。这在 Linux 上很简单，但在其他操作系统上，因为编译器（比如 macOS 使用的是 `clang`，而不是 `gcc`）和操作环境的其他方面的差异，可能就不那么简单了。

一旦获得了可以工作的构建版本，可以用如下开关激活 Shenandoah：

```
-XX:+UseShenandoahGC
```

Shenandoah 和其他收集器的暂停时间对比如图 7-9 所示。

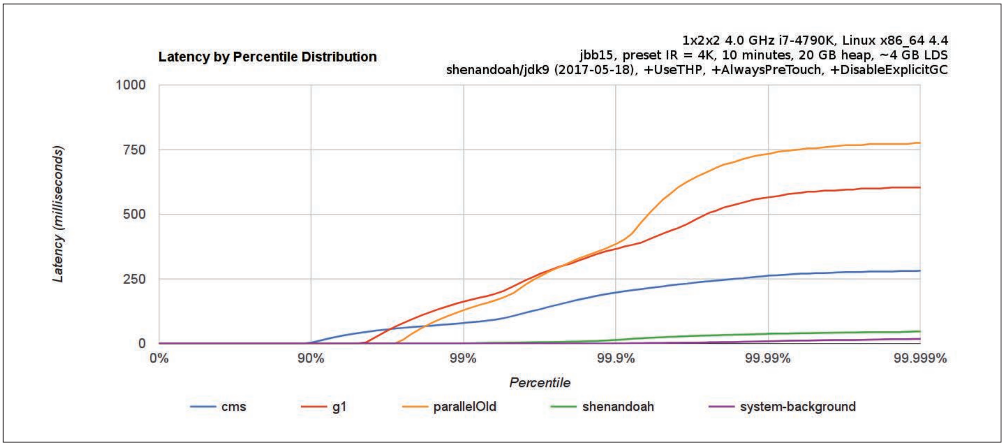


图 7-9：Shenandoah 与其他收集器的对比（Shipilëv）

人们还没有充分认识到 Shenandoah 不是一款分代收集器。随着它逐渐接近成为具有生产能力的收集器，这一设计决策的影响将变得更加清晰，但是对于性能敏感的应用程序而言，该收集器可能还是会让人担忧。

7.6 C4（Azul Zing）

Azul Systems 公司发布了两个不同的 Java 平台产品。其中一个是 Zulu，它是一个基于 OpenJDK 的 FOSS[7] 解决方案，支持多个平台。另一个是 Zing，一个只支持 Linux 的商用专有平台。虽然它使用的是来自 OpenJDK 的 Java 类库（遵守 Oracle 专有许可证），但使用了完全不同的虚拟机。



Zing 的一个重要的方面是，它从一开始就是为 64 位机器设计的，从来没有打算支持 32 位架构。

Zing 虚拟机包含了一些新颖的软件技术，包括 C4（continuously concurrent compacting collector）垃圾收集器和一些新颖的 JIT 技术，也包括 ReadyNow 和一个名为 Falcon 的编译器。

和 Shenandoah 一样，Zing 使用了一种并发压缩算法，但它没有利用 Brooks 指针。相反，Zing 使用的是单个 64 位字的对象头，而不是像 HotSpot 那样使用两个字的对象头。一个单独的对象头字中包含一个 kid，它是一个数字化的 klass ID（大约 25 位长），而不是 klass 指针。

图 7-10 中显示了 Zing 对象头，包括将某些 oop 引用位用作 LVB（loaded value barrie），而不是地址位。

注 7：FOSS：Free and Open Source Software，自由和开源软件。——译者注

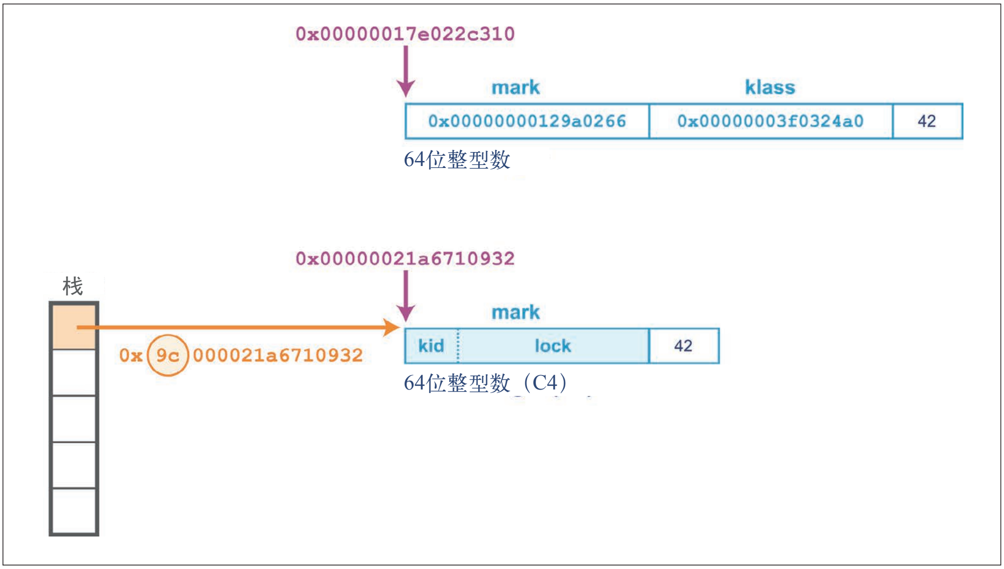


图 7-10：Zing 中的对象头布局

对象头底部的 32 位用于锁信息，这包括锁的状态以及跟锁的当前状态相关的额外信息。比如，在**轻量级锁**（thin lock）的情况下，这里存的就是拥有该锁的线程的 ID。关于轻量级锁的更多细节，参见 12.3 节。



因为 Zing 不支持压缩指针或同等技术，所以对于小于 30 GB 的堆，与同等的 HotSpot 堆相比，它的对象头更大，占用的堆空间也更多。

只支持 64 位架构的这种设计选择意味着 Zing 的元数据绝对不需要放到 32 位的大小中，也不需要分配扩展结构，再通过间接指针来寻址，从而避免了在 32 位 HotSpot 中的一些指针操作。

LVB

在 Shenandoah 收集器中，应用程序线程可以加载对可能已重新定位的对象的引用，并使用 Brooks 指针跟踪它们的新位置。LVB 的核心思想不是使用这种模式，而是提供一个解决方案，即一旦加载完成，每个已加载的引用会直接指向对象的当前位置。Azul 称之为**自愈屏障**（self-healing barrier）。

如果 Zing 跟随一个被收集器重新定位的对象的引用，那么在执行任何其他操作之前，应用程序线程将更新对对象新位置的引用，从而“愈合”了造成重新定位问题的原因。这意味着每个引用最多更新一次，如果该引用不再使用，就不做任何工作来保持更新。

除了对象头这个字之外，Zing 的对象引用（比如从栈上的某个局部变量指向保存在堆中的对象）会使用该引用的部分位来指示与对象垃圾收集状态相关的元数据。这是通过使用引

用本身的位而非一个单独的对象头字来节省一些空间。

Zing 定义了一个 `Reference` 结构，它是这样组织的：

```
struct Reference {
    unsigned inPageVA : 21;     // bits 0-20
    unsigned PageNumber: 21;    // bits 21-41
    unsigned NMT : 1;           // bit 42
    unsigned SpaceID : 2;       // bits 43-44
    unsigned unused : 19;       // bits 45-63
};

int Expected_NMT_Value[4] = {0, 0, 0, 0};

// Space ID values:
// 00 NULL and non-heap pointers
// 01 Old Generation references
// 10 New Generation references
// 11 Unused
```

`NMT`（not marked through）元数据位用于指示对象在当前收集周期中是否已经标记。C4 维护着一个目标状态，活对象都应该标记为该状态。当在标记过程中定位一个对象时，其 NMT 位会被设置为目标状态。因为在收集周期结束时，C4 会将目标状态位翻转，所以现在所有存活的对象都为下一个周期做好了准备。

C4 中的垃圾收集周期整体分为以下 3 个阶段。

1. 标记（mark）
2. 重新定位（relocate）
3. 重新映射（remap）

与 G1 一样，重新安置阶段将聚焦于最稀疏的 from 页面。这是意料之中的，因为 C4 是一款疏散型收集器。

C4 使用了一种叫作**交替压缩**（hand-over-hand compaction）的技术来提供连续压缩。这依赖于虚拟内存系统的一个特性——物理地址和虚拟地址是不连接的。在正常运行的情况下，虚拟内存子系统维护一个在进程地址空间中的虚拟页和底层的物理页之间的映射。

> HotSpot 没有使用系统调用来管理 Java 堆内存，与它不同的是，Zing 确实会在垃圾收集周期中通过调用进入内核。

通过使用 Zing 的疏散技术，对象被复制到一个不同的页面，这自然对应不同的物理地址，从而实现重新安置。一旦一个页面中的所有对象都完成复制，物理页面就可以释放并返还给操作系统。虽然仍会有应用程序中的引用指向现在未映射的虚拟页面，但是 LVB 会在内存故障发生之前处理并修正这些引用。

Zing 的 C4 收集器始终运行两个收集算法，一个针对新生代对象，另一个针对老年代对象。这显然是有开销的，但正如在第 8 章中研究的那样，当调优一款并发收集器（如 CMS）

时，假设收集器在运行时可能处于**背对背**模式对开销和容量规划非常有用，这与 C4 表现出的连续运行模式没有太大区别。

最终，性能工程师应该仔细研究转到 Zing 和 C4 收集器的收益和权衡。“测量，不要猜测”的理念不只适用于其他地方，也适用于虚拟机的选择。

7.7 IBM J9中的均衡收集器

IBM 开发了一款叫作 J9 的 JVM。它在历史上曾经是专有的，不过 IBM 正在将其开源，并重命名为 Open J9。这款虚拟机有几个不同的收集器，可以通过开关控制，其中包括一个与默认的并行收集器 HotSpot 类似的高吞吐量收集器。

本节将讨论均衡收集器（Balanced GC）。它是 64 位 J9 JVM 上的一款基于区域的收集器，是为大于 4 GB 的堆设计的。它的主要设计目标如下。

- 改进较大的 Java 堆上暂停时间的扩展性。
- 尽量减少最坏情况下的暂停时间。
- 利用对非均匀访存模型（non-uniform memory access，NUMA）的感知来提升性能。

为了实现第一个目标，堆被分割成若干个区域，对这些区域分别进行管理和收集。和 G1 一样，Balanced 收集器最多希望管理 2048 个区域，因此会选择一个区域大小来实现这一点。它的区域大小是 2 的幂，这点也和 G1 一样，但是 Balanced 收集器允许小到 512 KB 的区域。

正如我们对分代、基于区域的收集器的期望一样，每个区域都有一个与之关联的年龄，其中年龄为零的区域（Eden）用于分配新对象。当 Eden 空间满时，就必须执行一次收集。对此，IBM 的术语是**部分垃圾收集**（partial garbage collection，PGC）。

PGC 是一个 STW 的操作，它收集所有的 Eden 区域。如果收集器认为它们值得收集，那它也可以另外选择收集年龄较大的区域。这样看来，PGC 类似于 G1 的混合收集（Mixed GC）。

> 一旦 PGC 完成，包含存活对象的区域的年龄就会加 1，该区域有时被称为**代区域**。

与其他 J9 垃圾收集策略相比，另一个好处是类卸载可以增量执行。Balanced 可以在 PGC 期间收集属于当前收集集合的类加载器。与其他 J9 收集器不同，类加载器只能在全局收集期间收集。

Balanced 收集器的一个缺点是，因为 PGC 只能看到它所选择收集的区域，所以这种类型的收集可能会受到浮动垃圾的困扰。为了解决这个问题，Balanced 使用了一个**全局标记阶段**（global mark phase，GMP）。这是一个部分并发的操作，它会扫描整个 Java 堆，并标记可以收集的死对象。一旦 GMP 完成，随后的 PGC 就会对这些数据进行操作。因此，堆中浮动垃圾的数量会受到自上次 GMP 启动以来死亡的对象数量的限制。

Balanced 执行的最后一类垃圾收集操作是**全局垃圾收集**（global garbage collection，GGC），它是一个会压缩堆的完全 STW 的收集，类似于 HotSpot 中由并发模式失败而触发的 Full GC。

7.7.1 J9对象头

基本的 J9 对象头是一个 class 槽，其大小为 64 位，如果启用了压缩引用（compressed references），则为 32 位。



对于小于 57 GB 的堆，压缩引用默认开启，类似于 HotSpot 中的压缩指针技术。

但是，根据对象类型的不同，对象头可能会包含如下额外的槽：

- 同步对象将有 Monitor 槽；
- 放入内部 JVM 结构中的对象将有散列槽。

此外，Monitor 槽和散列槽不一定相邻于对象头，它们可以存储在对象中的任何地方，以便利用因对齐而浪费的空间。J9 的对象布局如图 7-11 所示。



图 7-11：J9 对象布局

class 槽的高 24 位（或 56 位）是一个指向 class 结构的指针，它位于堆外，类似于 Java 8 的 Metaspace。低 8 位是用于不同目的的标志，取决于所使用的垃圾收集策略。

7.7.2　Balanced收集器的大数组

在 Java 中分配大数组经常会触发压缩收集，因为必须找到足够的连续空间来满足分配。在讨论 CMS 时曾见过这种情况，即把空闲列表合并起来有时也不足以为一次大的分配释放足够空间，而且会导致并发模式失败。

对于一款基于区域的收集器，完全有可能遇到这样的情况：要分配的数组对象的大小超过了单个区域的大小。为了解决这个问题，Balanced 为大数组使用了另一种表示方式，就是允许将大数组分配到不连续的区块中。这种表示方式被称为 arraylet，它也是堆对象可以跨越不同区域的唯一情况。

arraylet 表示对用户的 Java 代码是不可见的，由 JVM 透明地处理。分配器把一个大数组当作一个中心对象，称为 spine。spine 中的条目会指向一些数组**叶子**，而这些叶子中包含真正的数组条目，从而使得读取数组条目只需要一次间接访问的成本。示例如图 7-12 所示。

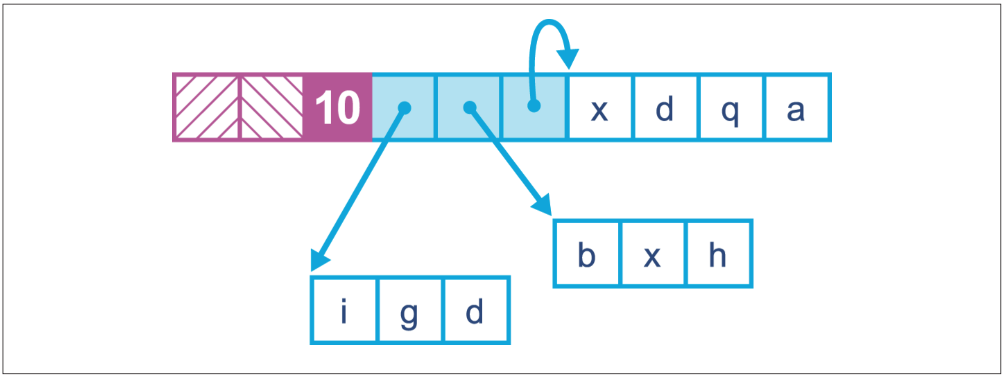


图 7-12：J9 中的 arraylet



尽管不能从普通的 Java 代码中看到 arraylet 表示，但是我们有可能通过 JNI API 看到它，所以程序员应该意识到，当从另一款 JVM 向当前平台移植 JNI 代码时，可能需要考虑 spine 和叶子的表示。

虽然在区域上执行 PGC 可以减少平均暂停时间，但是执行垃圾收集操作所花费的总时间可能会更多，因为维护区域之间的引用和被引用信息是有开销的。

最重要的是，需要进行全局 STW 收集或压缩（暂停时间的最坏情况）的可能性大大降低，而这通常是在堆满时能采取的最后手段。

由于管理区域和不连续的大数组是有开销的，因此 Balanced 收集器适用于那些更重视暂停时间而不是吞吐量的应用程序。

7.7.3　NUMA和Balanced收集器

NUMA 是用于多处理器系统（通常为中大型系统）中的一种内存架构。当处理器和内存被组织为节点时，这样的系统会涉及一个内存和处理器之间距离的概念。在一个给定的节点

上，处理器可以访问任何节点上的内存，但对本地内存（即内存和处理器属于同一节点）的访问要快得多。

对于跨多个 NUMA 节点执行的 JVM，均衡收集器可以在 NUMA 节点之间切割 Java 堆。在组织应用程序线程时，会考虑让它们便于在某个特定节点上执行，对象分配也会首选这个节点本地内存中的区域。这种安排的原理如图 7-13 所示。

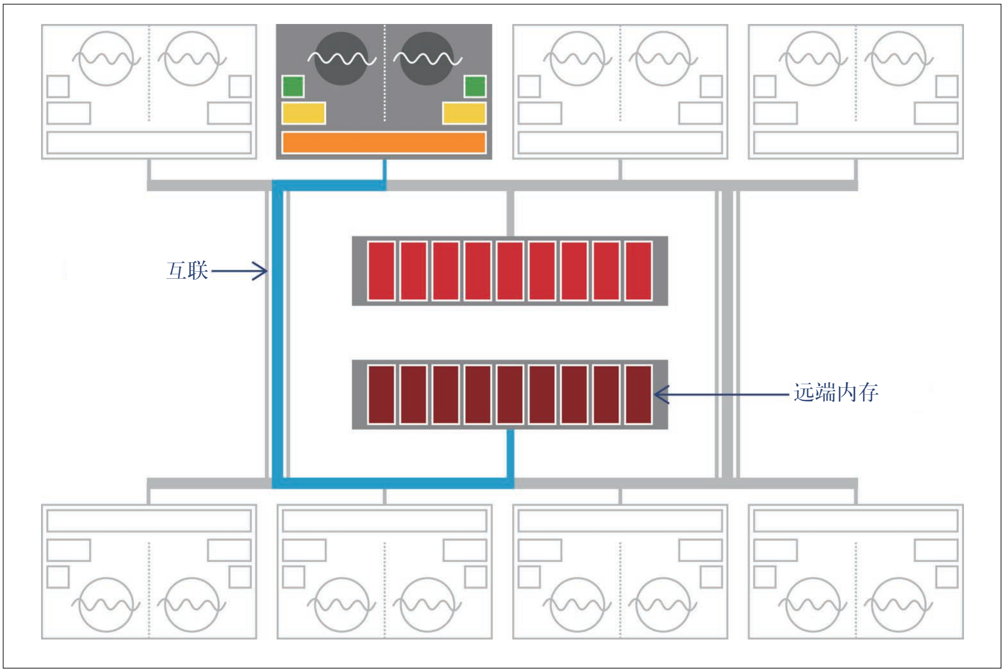


图 7-13：NUMA

此外，PGC 会尝试让对象靠近（内存距离）引用它们的对象和线程。这意味着线程引用的内存更有可能是本地的，从而可以提高性能。这个过程对应用程序不可见。

7.8 遗留的HotSpot收集器

在 HotSpot 的早期版本中，还有各种其他的收集器。出于完整性考虑，这里也介绍一下，但是其中的任何一个都不推荐在生产中使用，而且有些组合在 Java 8 中已经被废弃了，从 Java 9 开始将被禁用（或删除）。

7.8.1 Serial和SerialOld

Serial 和 SerialOld 收集器的运行方式与 Parallel GC 和 ParallelOld 收集器类似，但有一个重要的区别：它们只使用一个 CPU 核心来执行垃圾收集。尽管如此，它们也不是并发收集器，而且依然是完全 STW 的。当然，在现代多核系统上，使用这些收集器不会带来任何性能上的好处，所以不应该使用它们。

作为性能工程师，你应该了解这些收集器和它们的激活开关，以便确保当遇到设置了这些开关的应用程序时，你可以识别并删除它们。

这些收集器作为 Java 8 的一部分已被废弃，因此它们只会出现在仍在使用非常旧的 Java 版本的早期遗留应用程序中。

7.8.2 增量式CMS

增量式 CMS 收集器，一般被称为 iCMS，是对并发收集的较早尝试，它想引入到 CMS 中的一些想法也带来了后来的 G1。启用这种 CMS 模式的开关是：

```
-XX:+CMSIncrementalMode
```

有些专家仍然主张，在一些极端情况下（对于部署在只有一两个核心的旧式硬件上的应用程序），iCMS 从性能角度看可能是有效的选择。但是，几乎所有的现代服务器级应用程序都不应该使用 iCMS，而且它已经在 Java 9 中删除。



因为增量模式存在安全点行为和其他负面影响，所以除非有强有力的证据表明你的工作负载会从增量模式中受益，否则不应该使用它。

7.8.3 已被废弃和删除的垃圾收集组合

目前，通常废弃（deprecation）和删除（removal）的流程是，某些特性在一个 Java 版本中被标记为废弃，之后会在下一个版本或后来的版本中删除。相应地，在 Java 8 中，表 7-1 中的垃圾收集标志组合已经被标记为废弃且会在 Java 9 中删除。

表 7-1：被废弃的垃圾收集组合

组　合	标　记
DefNew + CMS	-XX:-UseParNewGC -XX:+UseConcMarkSweepGC
ParNew + SerialOld	-XX:+UseParNewGC
ParNew + iCMS	-Xincgc
ParNew + iCMS	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC
DefNew + iCMS	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC
CMS foreground	-XX:+UseCMSCompactAtFullCollection
CMS foreground	-XX:+CMSFullGCsBeforeCompaction
CMS foreground	-XX:+UseCMSCollectionPassing

在开始一次新的调优之前，你应该首先查看这个表，以便在一开始就能确定要调优的应用程序**没有**使用已被废弃的配置。

7.8.4 Epsilon

Epsilon 收集器并不是遗留的收集器，之所以把它放在这里，是因为**在任何情况下都不能将其应用于生产环境中**。但对于其他的收集器，如果在你的环境中遇到了，那么应立即将其

识别为极高风险，并标记为立即移除，而 Epsilon 略有不同。

Epsilon 是一款实验性的收集器，只为测试目的而设计。它不会执行任何实际操作来收集垃圾。运行 Epsilon 时所分配的每一个堆内存字节实际上都是一个内存泄漏，它们无法回收，最终会（可能会很快）导致 JVM 耗尽内存而崩溃。

> 开发一款只处理内存分配但不实现任何实际的内存回收机制的垃圾收集。一旦耗尽了可用的 Java 堆，则有序地关闭 JVM。
>
> ——Epsilon JEP

这样一款“收集器”对于下列目的非常有用：

- 性能测试和微基准测试；
- 回归测试；
- 测试低分配或零分配的 Java 应用程序或类库代码。

特别是，能够有把握地排除任何垃圾收集事件对性能数字的干扰对 JMH 测试很有帮助。内存分配回归测试如果能确保修改后的代码不会极大地改变分配行为，那它也会变得容易。开发人员还可以编写在 Epsilon 配置下运行的测试，该配置只接受有限数量的分配，当分配超过界限时，测试会因堆耗尽而失败。

最后，目前提议的虚拟机垃圾收集接口也将从 Epsilon 中受益，因为可以把它当作接口本身的一个最小测试用例。

7.9 小结

垃圾收集是 Java 性能分析和调优的一个基本方面。Java 丰富的垃圾收集器是这个平台的一大优势，但对于新手来说，它可能会让人望而生畏，特别是因为介绍每一种选择的权衡和性能结果的文档还比较少。

本章简要介绍了性能工程师所面临的决策，以及他们在决定为其应用程序选择合适的收集器时必须做出的权衡。我们讨论了一些基础理论，并介绍了一系列实现这些思想的现代垃圾收集算法。

下一章将把其中的一些理论运用到实际工作中，同时还会介绍日志、监控和工具，以使得关于垃圾收集性能调优的讨论更加科学严谨。

第 8 章

垃圾收集日志、监控、调优及工具

本章将介绍垃圾收集日志和监控这个宏大的主题。这是 Java 性能调优中最重要和最显著的一个方面，同时也最容易被误解。

8.1　认识垃圾收集日志

垃圾收集日志是一个重要的信息来源，对于与性能相关的一些悬而未决的案例分析特别有用，例如它提供了一些关于崩溃发生原因的见解。它使得分析人员甚至可以在没有活跃的应用程序进程可供诊断的情况下开始工作。

每一个严肃的应用程序都应该始终做到：

- 生成垃圾收集日志；
- 将其保存在一个与应用程序的输出独立的文件中。

对于生产型应用程序来说尤其如此。正如我们将看到的，垃圾收集日志并没有明显可观测到的开销，所以对于任何重要的 JVM 进程来说，它应该始终开启。

8.1.1　开启垃圾收集日志记录

首先要做的是在应用程序启动时添加一些开关。最好把这些开关看作必须打开的垃圾收集日志标志，任何 Java/JVM 应用程序（也许桌面应用程序可以除外）都应该启用。这些标志是：

```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintTenuringDistribution
-XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
```

接下来详细了解一下这些标志，其用法如表 8-1 所示。

表 8-1：必须打开的垃圾收集标志

标　　志	影　　响
-Xloggc:gc.log	控制垃圾收集事件写入哪个文件
-XX:+PrintGCDetails	将垃圾收集事件的详细信息写入日志
-XX:+PrintTenuringDistribution	添加对工具至关重要的垃圾收集事件额外细节
-XX:+PrintGCTimeStamps	打印垃圾收集事件的发生时间（自虚拟机启动后的秒数）
-XX:+PrintGCDateStamps	打印垃圾收集事件发生的挂钟时间

性能工程师应该注意到关于这些标志的以下一些细节。

- `PrintGCDetails` 标志取代了比较旧的 `verbose:gc` 标志。应用程序应该删除原来的标志。
- `PrintTenuringDistribution` 标志与其他标志不同，因为它提供的信息人很难直接处理，需要使用工具。该标志提供了计算关键的内存压力影响和事件（如过早晋升）所需的原始数据。
- `PrintGCDateStamps` 和 `PrintGCTimeStamps` 都是必需的，因为前者用于将垃圾收集事件与应用程序事件（在应用程序日志文件中）相关联，后者用于将垃圾收集和其他内部 JVM 事件相关联。

这种细节程度的日志记录不会对 JVM 的性能产生可测量的影响。当然，产生日志的量取决于许多因素，包括分配率、使用的收集器和堆大小（堆越小，则需要更频繁地进行垃圾收集，因此生成日志的速度也越快）。

为了让你有个大概的印象，我们来看个直观的例子，以第 6 章的模型分配器为例，每秒分配 50 MB 内存，运行 30 分钟，会产生约 600 KB 的日志。

除了必须使用的这些标志之外，还有一些控制垃圾收集的日志滚动的标志（如表 8-2 所示），许多应用程序支持团队会发现它们在生产环境中很有用。

表 8-2：垃圾收集日志滚动标志

标　　志	影　　响
-XX:+UseGCLogFileRotation	打开日志文件滚动
-XX:+NumberOfGCLogFiles=<n>	设置滚动日志文件的个数
-XX:+GCLogFileSize=<size>	设置滚动日志文件的最大值，当文件大小超过该参数值时，日志将写入下一个文件

要设置合理的日志滚动策略，应该与运维人员（包括 DevOps 人员）一起讨论决定。这种策略的选择以及对恰当的日志和工具的讨论不在本书范围之内。

8.1.2　垃圾收集日志与JMX的对比

我们在 2.7 节介绍了 VisualGC 工具，它能够实时显示 JVM 的堆状态。该工具实际上依靠 JMX（Java Management eXtension）接口来收集 JVM 的数据。关于 JMX 的完整讨论不在本书范围之内，但就 JMX 对垃圾收集的影响而言，性能工程师应该注意以下几点。

- 垃圾收集日志数据是由实际的垃圾收集事件驱动的，而 JMX 的来源数据是通过采样获得的。
- 垃圾收集日志数据获取的成本极低，而 JMX 存在隐含的代理和远程方法调用（remote method invocation，RMI）成本。
- 垃圾收集日志数据包含了与 Java 内存管理相关的性能数据的 50 多个方面，而 JMX 只有不到 10 个。

传统上，作为性能数据的来源，JMX 优于日志的一个方面是它可以提供开箱即用的流式数据。然而，诸如 jClarity Censum（参见 8.2 节）等现代工具也提供了将垃圾收集数据变为流式的 API，弥补了这一缺陷。

如果要对堆的基本使用情况进行粗略的趋势分析，那 JMX 是一个相当快速和简单的解决方案。然而，对于更深层次的问题诊断，它很快就力不从心了。

通过 JMX 获得的 Bean 是标准化的，而且很容易访问。VisualVM 工具提供了一种将该数据可视化显示的方式，市场上也有很多其他工具可用。

8.1.3　JMX的缺点

使用 JMX 监控应用程序的客户端，通常要依赖对运行时进行采样以获得当前状态的更新。为了连续获得数据，客户端需要轮询运行时中的 JMX Bean。

在存在垃圾收集的情况下，这会导致一个问题：客户端无法知道收集器何时运行，进而也意味着每个收集周期前后的内存状态都是未知的。因此，我们无法对垃圾收集数据执行一系列更深入和更精确的分析技术。

即便如此，基于 JMX 数据的分析仍然有用，但仅限于确定长期趋势。然而，如果想准确地调优某个垃圾收集器，我们需要做得更好。特别是能够了解每次收集前后堆的状态是非常有用的。

此外，还有一组围绕内存压力的极其重要的分析（即分配率）也因为从 JMX 收集数据的这种方式而不再能够执行。

不仅如此，`JMXConnector` 规范目前的实现依赖于 RMI。因此，使用 RMI 通信通道会遇到的任何问题，使用 JMX 也同样会遇到。具体包括：

- 打开防火墙中的端口，以便可以建立后续的套接字连接；
- 使用代理对象以方便调用 `remove()` 方法；
- 依赖于 Java 终结化（finalization）。

对于少数 RMI 连接来说，关闭连接所需的工作量微乎其微。然而，清理工作要依赖于终结化。这意味着必须运行垃圾收集器以回收该对象。

JMX 连接的生命周期这一性质在大多数情况下会导致 RMI 对象直到一次 Full GC 时才会被收集。关于终结化的影响以及为什么应该避免终结化的更多细节参见 11.6 节。

默认情况下，任何使用 RMI 的应用程序每小时都会触发 Full GC。对于已经使用 RMI 的应用程序，使用 JMX 不会增加成本。但是，对于还没有使用 RMI 的应用程序，如果决定使用 JMX，必然存在额外的影响。

8.1.4 垃圾收集日志数据带来的好处

现代的垃圾收集器包含了很多不同的活动部件，将其组合在一起得到的是一个非常复杂的实现。实现如此复杂，以致某个收集器的性能即使不是不可能的，也是很难预测的。这些类型的软件系统就是所谓的**浮现式的**（emergent），因为它们的最终行为和性能是所有组件如何共同工作和执行的结果。不同的压力会以不同方式影响不同的组件，从而导致成本模型也会动态发生变化。

最初，Java 垃圾收集的开发人员添加了垃圾收集日志来帮助调试其实现，因此，60 个左右的垃圾收集相关标志所产生的数据中，有很大一部分是出于性能调试的目的。

随着时间的推移，负责对应用程序中的垃圾收集过程进行调优的人开始意识到，鉴于垃圾收集调优的复杂性，对运行时中正在发生的事情有个精确的了解也会让他们受益匪浅。因此，不管采用何种调优方式，能够收集和阅读垃圾收集日志都是很有帮助的。

> 垃圾收集日志是在 HotSpot JVM 内部使用非阻塞写入机制来完成的。它对应用程序的性能没有影响，所有生产环境中的应用程序都应该开启垃圾收集日志。

由于垃圾收集日志中的原始数据都可以与特定的垃圾收集事件关联在一起，因此我们可以对其进行各种有用的分析，从而了解垃圾收集的开销，进而知道哪些调优动作更有可能产生正面的结果。

8.2 日志解析工具

不同于语言规范和虚拟机规范，垃圾收集日志消息并没有一个标准的格式。HotSpot 垃圾收集开发团队可以自行决定任何一条消息的内容。不同的小版本之间，格式都有可能发生改变，而且确实存在改变的情况。

虽然最简单的日志格式很容易解析，但随着垃圾收集日志标志的添加，所产生的日志输出就变得更为复杂了，因而情况也就进一步复杂化。并发收集器生成的日志更是如此。

经常有这样的情况发生，即系统采用了手写的垃圾收集日志解析器，而有人修改了垃圾收集配置，这使得日志输出格式发生了变化，从而导致在此之后的某个时间点，解析器不能继续工作。调查其原因，矛头指向了垃圾收集日志，团队发现自己开发的解析器无法处理改变后的日志格式——解析器在日志信息最有价值的那个点停止了工作。

> 不建议开发人员自己解析垃圾收集日志。相反，应该使用某个工具。

本节将研究两个还在活跃维护的工具，一个是商业的，一个是开源的。社区也有其他一些工具（比如 GarbageCat），不过都是偶尔维护，或者根本不维护了。

8.2.1 Censum

Censum 内存分析器是由 jClarity 开发的商业工具，它既可以作为桌面工具（用于单个 JVM 的手动分析），也可以作为监控服务（用于大规模 JVM 集群）。该工具致力于提供可用性最好的垃圾收集日志解析、信息提取和自动分析功能。

图 8-1 中的 Censum 桌面视图显示的是运行 G1 垃圾收集器的金融交易应用程序的分配率。即使从这个简单的视图中，我们也可以看到交易应用程序中有几个时间段分配率很低，而且正好与市场的平静期相一致。

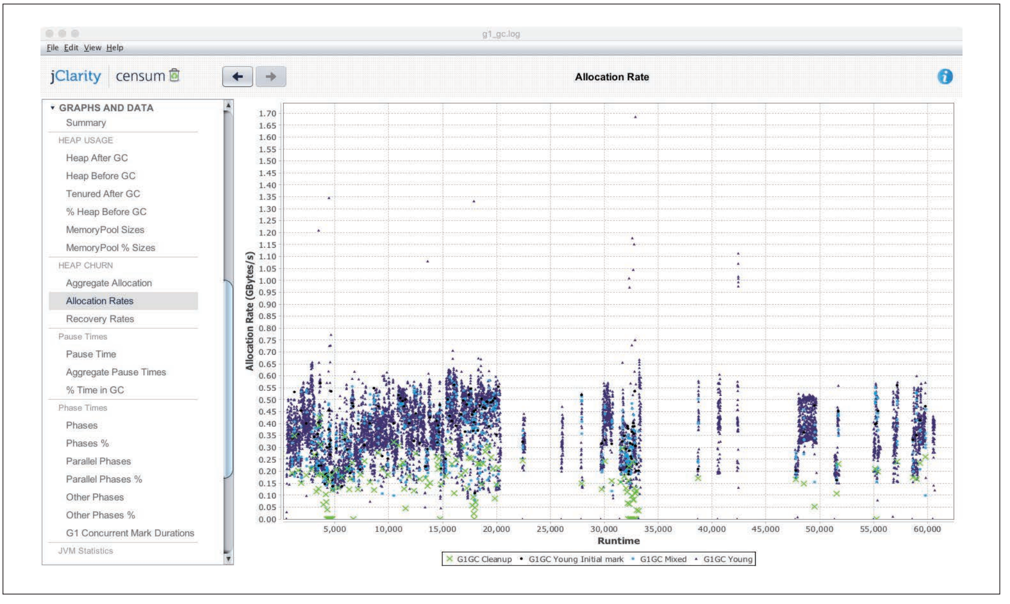


图 8-1：Censum 分配视图

Censum 提供的其他视图还包括暂停时间图，如来自 SaaS 版本的图 8-2 所示。

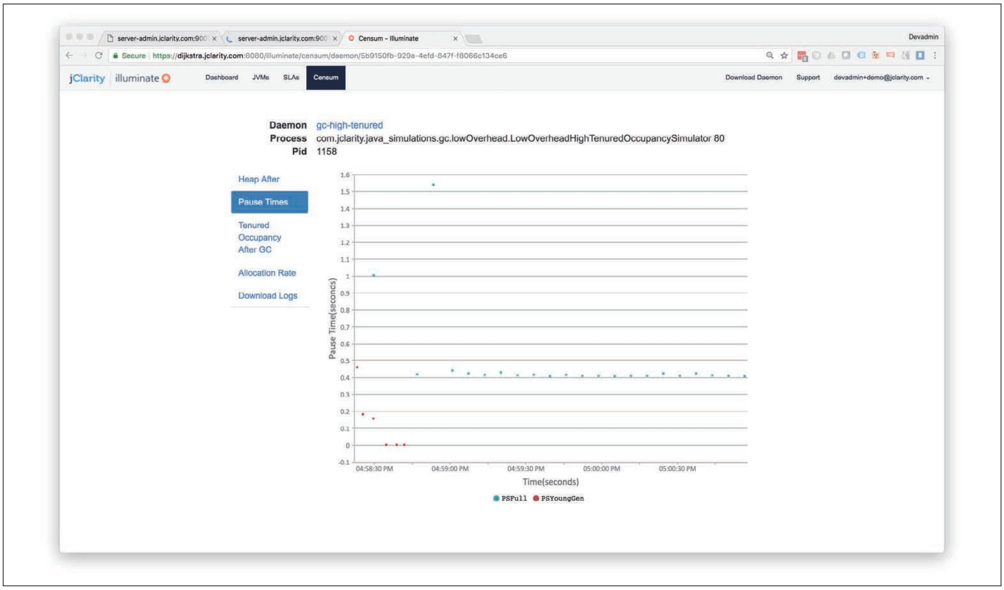


图 8-2：Censum 的暂停时间视图

使用 Censum SaaS 监控服务的一个优势是能够同时查看整个集群的健康状况，如图 8-3 所示。对于正在进行的监控，这通常要比每次处理一个 JVM 更容易。

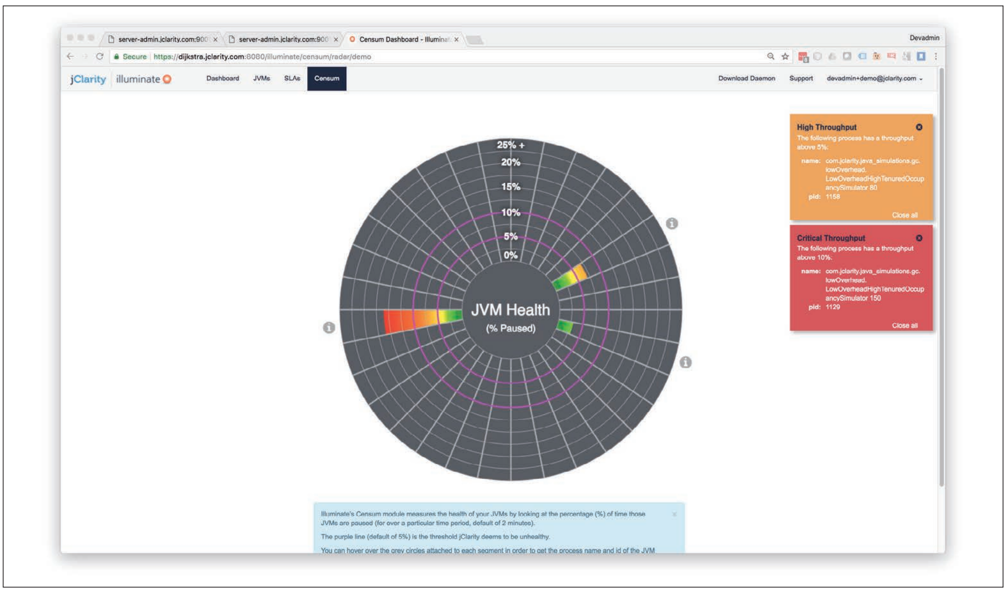


图 8-3：Censum 集群健康状况概览

Censum 在开发过程中密切关注日志文件的格式，并对所有可能影响日志的 OpenJDK 的代码签入进行监控。Censum 支持从 1.4.2 到当前所有版本的 Sun/Oracle Java 以及所有的收集

器，它能支持的垃圾收集日志配置的数量也是市面上所有工具中最多的。

从目前的版本来看，Censum 支持对以下这些方面进行自动分析：

- 准确的分配率
- 过早晋升
- 峰值分配
- 暂停时间
- 内存泄漏检测
- 堆大小和容量规划
- 操作系统对虚拟机的干扰
- 内存池大小是否合适

关于 Censum 和试用许可证的更多详情，请访问 jClarity 网站。

8.2.2　GCViewer

GCViewer 是一个桌面工具，提供了一些基本的垃圾收集日志解析和图形化显示功能，其最大的优点在于它是开源软件，可以免费使用。然而，它提供的功能没有商业工具那么多。

要使用 GCViewer，需要下载源代码。在编译和构建后，可以将其打包成一个可执行的 JAR 文件。

然后可以在 GCViewer 主界面中打开垃圾收集日志文件，示例如图 8-4 所示。

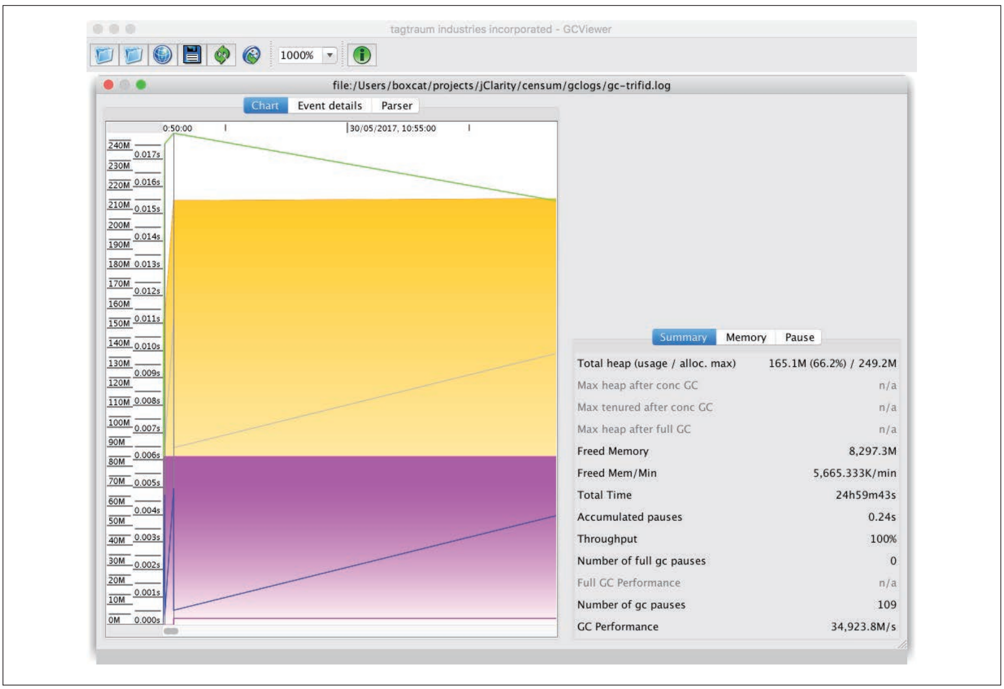


图 8-4：GCViewer

GCViewer 缺乏分析能力，对于 HotSpot 可能生成的各种垃圾收集日志格式，它只能解析一部分。

可以将 GCViewer 用作解析库，将数据点导出到一个可以可视化显示的工具中，但这需要在现有开源代码的基础上进行额外的开发。

8.2.3　对于同一数据的不同可视化效果

你应该知道，对于同样的数据，不同的工具可能会生成不同的可视化效果。我们在 6.6 节见过的简单锯齿模式，就是建立在一个以堆的整体大小作为观测量的采样视图之上的。

使用 GCViewer 的“垃圾收集后的堆占用情况”（Heap Occupancy after GC）视图将产生该模式的垃圾收集日志绘制成图，我们就得到了如图 8-5 所示的可视化效果。

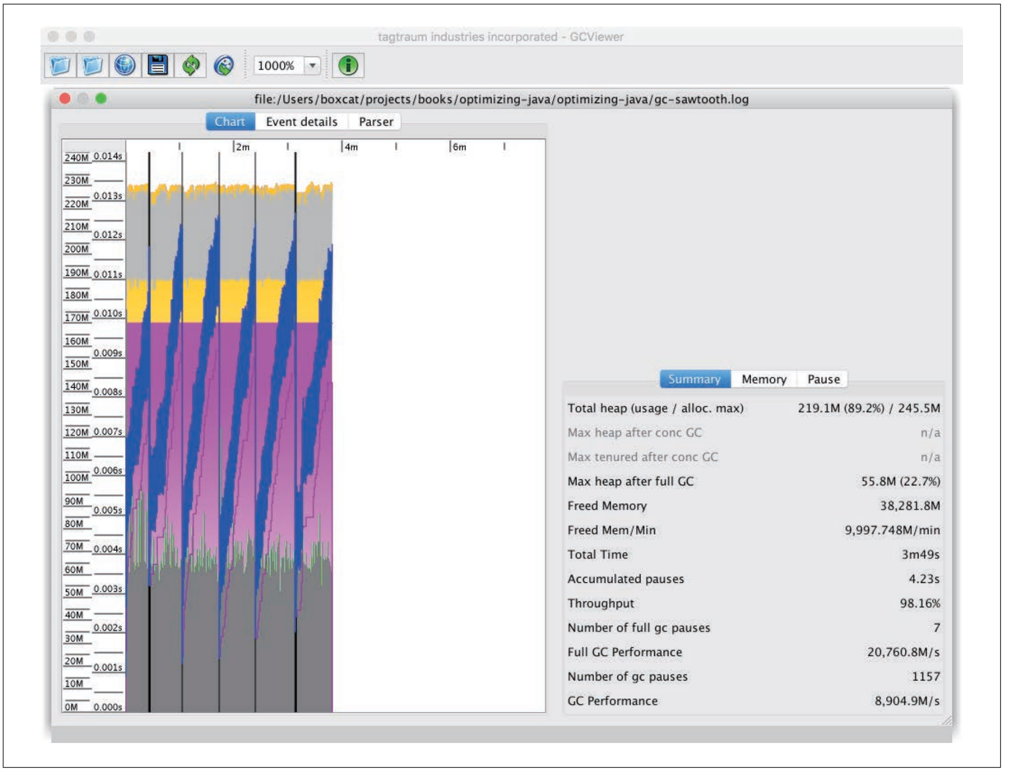


图 8-5：GCViewer 中的简单锯齿模式

下面再来看看同样的简单锯齿模式在 Census 中的显示，如图 8-6 所示。

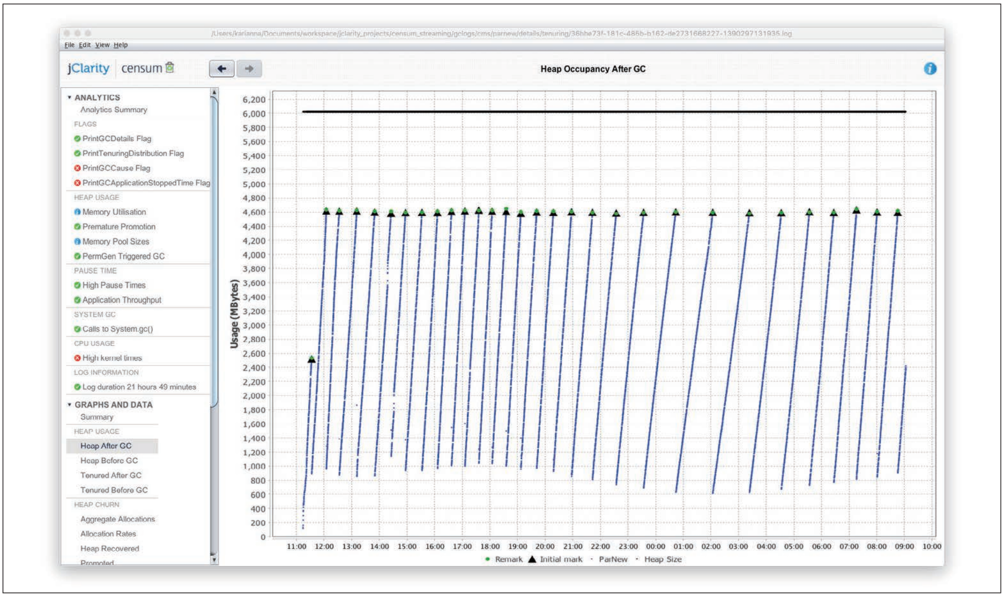


图 8-6：Censum 中的简单锯齿模式

虽然图像和可视化效果不同，但在这两种情况下工具给出的信息是一样的——堆正在正常运行。

8.3 基本垃圾收集调优

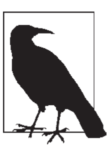
当工程师考虑 JVM 调优策略时，经常会出现这样一个问题：“我应该何时对垃圾收集进行调优呢？”与其他任何调优技术一样，垃圾收集调优应该是整个诊断过程的一部分。记住关于垃圾收集调优的以下事实非常有用。

1. 排除或确认垃圾收集是性能问题的根源所付出的成本很低。
2. 在用户验收测试中开启垃圾收集标志的成本很低。
3. 设置内存剖析器或执行剖析器的成本不低。

工程师还应该知道在调优过程中要研究和测量以下 4 个主要因素：

- 分配
- 对暂停的灵敏度
- 吞吐量行为
- 对象生命周期

其中分配往往是最重要的。



吞吐量会受到很多因素的影响，比如并发收集器在运行的时候也会占用处理器核心。

现在来看看表 8-3 中列出的一些基本的设置堆大小的标志。

表8-3：设置GC堆大小的标志

标　　志	影　　响
-Xms<size>	设置为堆保留的内存的最小值
-Xms<size>	设置为堆保留的内存的最大值
-XX:MaxPermSize=<size>	设置 PermGen 允许的最大值（Java 7）
-XX:MaxMetaspaceSize=<size>	设置 Metaspace 允许的最大值（Java 8）

MaxPermSize 标志是之前版本遗留的，只适用于 Java 7 及之前的版本。在 Java 8 及之后的版本中，PermGen 已经被移除并被 Metaspace 取代。

> 如果在一个 Java 8 应用程序中设置 MaxPermSize，那么应该立刻删除该标志。因为 JVM 会忽略它，所以很明显它不会对应用程序产生任何影响。

当涉及使用额外的垃圾收集标志进行调优时，应该：

- 每次只添加一个标志；
- 确保你完全理解每个标志的效果；
- 记得有些组合会产生副作用。

检查垃圾收集是否是性能问题的原因相对来说比较容易，就是假设事件是实时发生的。第一步是使用 vmstat 或类似工具查看机器的高级指标，如 3.6 节所讨论的那样。首先，登录到存在性能问题的机器，检查是否有如下问题：

- CPU 利用率接近 100%；
- 这个时间的绝大部分（90%以上）是在用户空间中消耗的；
- 垃圾收集日志显示有活动，表明垃圾收集目前正在运行。

这里假设的是问题正在发生，而且工程师可以实时观测到。对于过去的事件，必须有足够的历史监控数据（包括 CPU 利用率和有时间戳的垃圾收集日志）。

如果以上 3 个条件都满足，则应将垃圾收集作为当前性能问题的最可能的原因进行调查和调优。这个测试非常简单，而且有一个很明确的结果——要么“垃圾收集正常”，要么“垃圾收集不正常”。

如果发现垃圾收集是性能问题的根源，那么下一步就是了解分配和暂停时间行为，然后对垃圾收集进行调优，必要时可能还需要内存剖析器。

8.3.1　理解分配行为

对于确定如何调优垃圾收集器，以及是否能实际通过调优垃圾收集器来从根本上提升性能，分配率分析都是非常重要的。

可以使用新生代收集事件中的数据来计算分配的数据量和两次收集之间的时间，然后利用这些信息计算该时间间隔内的平均分配率。

与其花费时间和精力手动计算分配率，不如使用工具来计算这个数字。

经验表明，持续超过 1 GB/s 的分配率几乎总是表明存在性能问题，而且这些问题是无法通过调优垃圾收集器来解决的。在这种情况下，提高性能的唯一方法是通过重构来消除应用程序关键部分中的内存分配，从而改进内存使用效率。

我们可以将 VisualVM（参见 2.7 节）甚至 `jmap`（参见 6.1 节）所生成的简单的内存直方图作为理解内存分配行为的起点。一个有用的初始分配策略专注于以下 4 个简单的方面：

- 不重要的、可避免的对象分配（例如日志调试消息）
- 装箱开销
- 领域对象
- 大量的非 JDK 框架对象

对于第一个方面，只需要发现并删除不必要的对象创建。大量的装箱操作可能是其中的一种形式，但是也有很多其他例子是浪费对象创建的可能来源，比如自动生成的用于序列化 / 反序列化为 JSON 的代码，或者 ORM 代码。

领域对象通常不是应用程序内存利用率的主要贡献者，更常见的是以下类型：

- `char[]`：组成字符串的字符
- `byte[]`：原始二进制数据
- `double[]`：计算数据
- Map 条目
- `Object[]`
- 内部数据结构（比如 `methodOop` 和 `klassOop`）

只需要一个简单的直方图，往往就可以发现内存泄漏或大量创建了不必要的领域对象等问题，这仅仅是因为它们会出现于堆直方图的顶部元素中。通常，所需要做的就是快速计算出领域对象的预期数据量，看看观测到的数据量是否符合预期。

我们在 6.4.1 节见过线程本地分配，该技术的目的是为每个线程提供一个私有的区域来分配新对象，从而实现 $O(1)$ 分配。

TLAB 的大小会根据每个线程的情况进行动态调整，如果有空间，那常规对象就会在 TLAB 中分配。如果没有，线程则会向虚拟机请求一个新的 TLAB 并再次尝试分配。

如果这个对象连空的 TLAB 都容纳不下，那虚拟机接下来就不会考虑在 TLAB 中分配对象，而是尝试想办法在 Eden 区中直接分配对象。如果还是失败，下一步就是执行一次 Young GC（这可能会调整堆的大小）。如果仍是失败，依然没有足够的空间，那最后的办法就是直接在 Tenured 区中分配该对象。

由此可知，最终真正有可能直接分配在 Tenured 区的对象只有大数组（尤其是字节和字符数组）。

HotSpot 有几个与 TLAB 和大对象提前进入 Tenured 区有关的调优标志：

```
-XX:PretenureSizeThreshold=<n>
-XX:MinTLABSize=<n>
```

和所有的开关一样，在没有基准测试和可靠的证据能证明其确实有效之前，不应该使用它们。在大多数情况下，内置的动态行为就会产生很好的结果，而任何修改都不会产生什么明显的影响。

分配率也会影响晋升到 Tenured 区的对象数量。如果我们假设寿命短暂的 Java 对象有一个固定的生命周期（用挂钟时间表示），那么较高的分配率将导致多次执行 Young GC 的时间间隔更近。如果收集太过频繁，那么寿命短暂的对象可能还没来得及死亡就被错误地晋升到 Tenured 区。

换句话说，分配的高峰会导致出现我们在 6.6 节见过的过早晋升问题。为防止这种情况发生，JVM 会动态调整 Survivor 空间的大小以容纳更多存活下来的数据，而不用将其晋升到 Tenured 区。

下面这个 JVM 开关有时对处理提前进入 Tenured 区的问题和过早晋升有所帮助：

```
-XX:MaxTenuringThreshold=<n>
```

它可以控制一个对象在晋升到 Tenured 区之前必须经历的垃圾收集次数，其默认值是 4，但是可以将其设置为 1 到 15 之间的任何值。修改这个值代表了在如下两个考虑之间的权衡：

- 阈值越高，真正长寿的对象被复制的次数就越多；
- 阈值太低，一些寿命短暂的对象就会被晋升，进而增加 Tenured 区的内存压力。

阈值太低的一个可能后果是，由于更多的对象被晋升到 Tenured 区，导致该区域被更快填满，因此 Full GC 就会更频繁地发生。和以前一样，如果没有明确的基准测试表明将某个开关修改为非默认值可以提升性能，那就不要改动该开关。

8.3.2 理解暂停时间

开发人员经常会对暂停时间产生认知偏差。很多应用程序可以轻松容忍超过 100 毫秒的暂停时间。人眼每秒只能对单个数据项进行 5 次更新，所以对于大多数需要和人打交道的应用程序（如 Web 应用）而言，我们看不到 100~200 毫秒的暂停时间。

对于暂停时间的调优，一个有用的启发式方法是将应用程序分为 3 个大的区间。这些区间基于应用程序对响应的需求来划分，表现为应用程序可以容忍的暂停时间。

1. 大于 1 秒：可以容忍超过 1 秒的暂停时间。
2. 100 毫秒 ~ 1 秒：可以容忍超过 100 毫秒但小于 1 秒的暂停时间。
3. 小于 100 毫秒：不能容忍超过 100 毫秒的暂停时间。

如果将应用程序对暂停的灵敏度与其预期的堆大小结合起来考虑，那我们就可以构建一个最佳评估表来判断应该选择哪种收集器，结果如表 8-4 所示。

表8-4：初始收集器选择

可以容忍的暂停时间			堆　大　小
大于 1 秒	100 毫秒 ~ 1 秒	小于 100 毫秒	小于 2 GB
Parallel	Parallel	CMS	小于 4 GB
Parallel	Parallel/G1	CMS	小于 4 GB
Parallel	Parallel/G1	CMS	小于 10 GB
Parallel/G1	Parallel/G1	CMS	小于 20 GB
Parallel/G1	G1	CMS	大于 20 GB

希望这些指南和经验法则成为调优的起点，但它们并非是完全明确的规则。

展望未来，随着 G1 收集器不断成熟，我们有理由期待它会扩展到支持目前 ParallelOld 所涵盖的更多用例。它也有可能扩展到支持 CMS 的用例，不过可能性不是很大。

当使用并发收集器时，在尝试对暂停时间进行调优之前，仍然应该减少分配。减少分配也会减少并发收集器面对的内存压力，从而使收集周期更容易跟上分配线程的速度。这将降低出现并发模式失败的可能性，而对暂停敏感的应用程序来说，则需要尽可能避免出现并发模式失败这样的事件。

8.3.3　收集器线程和GC根

一个有用的思维训练是“像垃圾收集线程一样思考”，这可以让我们深入了解收集器在各种情况下的行为。然而，与垃圾收集的其他很多方面一样，总是存在一些基本的权衡。这些权衡包括定位 GC 根所需的扫描时间会受到以下因素的影响：

- 应用线程的数量
- 代码缓存中已编译代码的数量
- 堆的大小

即使对于垃圾收集的这一个方面来讲，以上哪个因素会对 GC 根的扫描起主导作用也总是取决于运行时条件和可以并行化的程度。

例如，考虑在标记阶段发现一个巨型 `Object[]` 的情况。因为扫描将由单个线程完成，所以不会存在工作窃取。在极端情况下，单线程扫描时间将占据整个标记时间。

事实上，对象图越复杂，这种效果就越明显。这意味着图中存在的对象“长链”越多，标记时间就会越差。

如果应用程序线程的数量很多，那也会对垃圾收集时间产生影响，因为这代表着有更多的栈帧需要扫描，并且达到某个安全点也需要更多时间，而且不管是在裸机还是虚拟化环境中，它们还会给线程调度器带来更大的压力。

GC 根除了这些传统的例子之外，还有其他来源，包括 JNI 帧和被 JIT 编译的代码的代码缓存（参见 9.4 节）。

在代码缓存中的 GC 根扫描是单线程的（至少对于 Java 8 来说是这样）。

在这 3 个因素中，栈和堆扫描的并行化程度相当高。分代收集器还会记录来自其他内存池的根，在 G1 中会使用像 RSet 这样的机制，而在 Parallel GC 和 CMS 中则使用卡表。

例如，让我们来考虑 6.3 节中介绍过的卡表。它们被用于指示包含从老年代指向新生代的引用的内存块。因为卡表中的 1 字节代表老年代中的 512 字节，所以很明显，如果老年代内存有 1 GB，则必须扫描 2 MB 大小的卡表。

为了解扫描一个卡表需要多长时间，我们来看一个简单的基准测试，模拟扫描 20 GB 大小的堆的卡表：

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(1)
public class SimulateCardTable {

    // 老年代占了堆的3/4，对于1GB的老年代空间，需要2MB大小的卡表
    private static final int SIZE_FOR_20_GIG_HEAP = 15 * 2 * 1024 * 1024;

    private static final byte[] cards = new byte[SIZE_FOR_20_GIG_HEAP];

    @Setup
    public static final void setup() {
        final Random r = new Random(System.nanoTime());
        for (int i=0; i<100_000; i++) {
            cards[r.nextInt(SIZE_FOR_20_GIG_HEAP)] = 1;
        }
    }

    @Benchmark
    public int scanCardTable() {
        int found = 0;
        for (int i=0; i<SIZE_FOR_20_GIG_HEAP; i++) {
            if (cards[i] > 0)
                found++;
        }
        return found;
    }
}
```

运行这个基准测试，会看到类似于下面的输出：

```
Result "scanCardTable":
  108.904 ±(99.9%) 16.147 ops/s [Average]
  (min, avg, max) = (102.915, 108.904, 114.266), stdev = 4.193
  CI (99.9%): [92.757, 125.051] (assumes normal distribution)
```

```
# Run complete. Total time: 00:01:46

Benchmark                          Mode  Cnt    Score    Error  Units
SimulateCardTable.scanCardTable   thrpt    5  108.904 ± 16.147  ops/s
```

这表明对于一个 20 GB 大小的堆，扫描其卡表大约需要 10 毫秒。当然，这是单线程扫描的结果。但是，我们据此对新生代收集的暂停时间有了一个有用的大致下限。

前面介绍了一些可用于调优大部分收集器的通用技术，下面我们来看一些与具体收集器相关的调优方法。

8.4 调优Parallel GC

Parallel GC（并行垃圾收集器）是最简单的收集器，因此它也是最容易调优的就并不奇怪了，不过它通常只需要最少的调优。Parallel GC 的目标和取舍很清晰：

- 完全 STW；
- 垃圾收集吞吐量高 / 计算成本低；
- 不可能出现部分收集；
- 暂停时间会随堆大小的增加线性增长。

如果应用程序能够忍受 Parallel GC 的特性，那么它可能是非常有效的选择——特别是在小堆上，比如那些小于 4 GB 的堆。

较老的应用程序可能使用了显式设置大小的标志来控制各个内存池的相对大小，这些标志汇总在表 8-5 中。

表8-5：较老的设置垃圾收集堆大小的标志

标　　志	影　　响
-XX:NewRatio=<n>	（旧标志）设置老年代与新生代的相对比例
-XX:SurvivorRatio=<n>	（旧标志）设置 Survivor 空间占新生代的比例
-XX:NewSize=<n>	（旧标志）设置新生代的最小值
-XX:MaxNewSize=<n>	（旧标志）设置新生代的最大值
-XX:MinHeapFreeRatio	（旧标志）设置堆空间最小空闲比例；当堆空间的空闲内存小于这个数值时，扩展堆空间
-XX:MaxHeapFreeRatio	（旧标志）设置堆空间最大空闲比例；当堆空间的空闲内存大于这个数值时，压缩堆空间

Survivor 空间比例、新生代空间比例和堆的整体大小之间通过下面的公式连接起来：

```
Flags set:

-XX:NewRatio=N
-XX:SurvivorRatio=K

YoungGen = 1 / (N+1) of heap
OldGen = N / (N+1) of heap
```

```
Eden = (K - 2) / K of YoungGen
Survivor1 = 1 / K of YoungGen
Survivor2 = 1 / K of YoungGen
```

对于大多数现代应用程序而言，不应该使用这类显式设置大小的操作，因为 JVM 的自动优化几乎在所有情况下比人做得好。对于 Parallel GC 来说，实在没有别的办法的时候才可以考虑这些开关。

8.5 调优CMS

CMS 收集器素有难以调优之名，这并非完全没有道理，因为要想让 CMS 发挥出最佳性能，其中的复杂性和权衡不容小觑。

不幸的是，许多开发人员简单地认为“暂停时间不好，所以并发标记收集器好”。像 CMS 这样的低暂停收集器实际应该被视为最后的手段，只有在用例真正需要低 STW 暂停时间的情况下才使用。否则，团队可能会被一个难以调优的收集器所困扰，而且这对应用程序的性能也没有任何实际的好处。

CMS 有大量的标志（截至 Java 8u131，几乎达到 100 个），一些开发人员可能会受到诱惑，试图通过修改这些标志的值来提高性能。然而，这很容易导致出现我们在第 4 章中遇到的一些反模式，包括：

- 摆弄开关
- 按照坊间传说调优
- 忽略大局

认真的性能工程师应该抵制这种诱惑，以免陷入这些认知陷阱。



对于大多数使用 CMS 的应用程序而言，可能并不能通过修改 CMS 命令行标志的值就获得真正可见的性能改进。

尽管有这样的危险，但在某些情况下，仍然需要通过调优来改进 CMS 性能，或者使其性能可以接受。我们先来考虑一下吞吐量行为。

当一次 CMS 收集运行时，默认情况下有一半的处理器核心在处理垃圾收集。这不可避免地会导致应用程序的吞吐量降低。一个有用的经验法则是考虑收集器在并发模式失败之前的情况。

在这种情况下，只要 CMS 收集一完成，就会立即启动新的 CMS 收集，这称为**背靠背**收集，在并发收集器的情况下，它预示着并发收集器即将崩溃的时间点。如果应用程序分配加快，那么回收将无法跟上，进而导致并发模式失败。

在背靠背收集的情况下，基本上整个应用程序运行的吞吐量将减少 50%。在进行调优时，工程师应该考虑应用程序是否能够容忍这种最坏的情况。如果不能，那可能就应该将应用程序迁移到拥有更多处理器核心可用的主机上运行。

另一种方法是在 CMS 周期内减少分配给垃圾收集的核心数量。当然，这是很危险的，因为它减少了执行收集可用的 CPU 时间，从而降低了应用程序对分配峰值的弹性（反过来可能会使它更容易受到并发模式失败的影响）。控制这个问题的开关是：

```
-XX:ConcGCThreads=<n>
```

很明显，如果一个应用程序在默认设置下无法快速回收，那么减少垃圾收集线程的数量只会让事情变得更糟。

CMS 有以下两个独立的 STW 阶段。

初始标记阶段

标记出紧邻的内部节点——由 GC 根直接指向的节点。

重新标记

使用卡表来识别可能需要修复的对象。

这意味着所有的应用程序线程都必须停止，因此每个 CMS 周期会两次进入安全点。对于某些对安全点行为非常敏感的低延迟应用而言，其产生的影响可能非常明显。

有时一起出现的两个标志是：

```
-XX:CMSInitiatingOccupancyFraction=<n>
-XX:+UseCMSInitiatingOccupancyOnly
```

这些标志可能非常有用，它们还再次说明了分配率不稳定所带来的重要影响，以及由之而来的两难问题。

`CMSInitiatingOccupancyFraction` 标志用于确定 CMS 应该在何时开始执行收集。当 CMS 运行的时候，堆中需要有一些净空间，以便当出现对象从新生代晋升而来时，有空间可用。

像 HotSpot 实现垃圾收集的方法的其他许多方面一样，这个空余空间的大小由 JVM 本身收集的统计数据控制。然而，对于 CMS 的第一次运行而言，仍然需要一个估算值。这个初始的估算值就是由标志 `CMSInitiatingOccupancyFraction` 来控制的，其默认值为 75%，它意味着当堆的使用达到这个比例时，将触发第一次 CMS Full GC。

如果也设置了 `UseCMSInitiatingOccupancyOnly` 标志，则占用比例不会再动态调整。不要轻易这样做，因为在实践中净空间一般不会减少（也就是将参数值增加到 75% 以上）。

然而，对于一些分配率非常高的 CMS 应用程序来说，一种策略是在关闭自适应大小的同时增加净空间（降低参数值）。这样做的目的是试图减少并发模式失败，但代价是会增加 CMS 并发垃圾收集的发生频率。

由碎片化导致的并发模式失败

现在来看看执行调优分析所需要的数据只能在垃圾收集日志中获得的情况。在这种情况下，我们想使用**空闲链表统计**来预测 JVM 可能在何时会因为堆的碎片化而遭受 CMF。这种类型的 CMF 是由 CMS 维护的空闲链表引起的，我们在 7.3.1 节遇到过。

需要打开另一个 JVM 开关才能看到额外的输出：

```
-XX:PrintFLSStatistics=1
```

在垃圾收集日志中产生的输出是这样的（这里演示的是 `BinaryTreeDictionary` 基准测试的输出统计细节）：

```
Total Free Space: 40115394

Max Chunk Size: 38808526

Number of Blocks: 1360

Av. Block Size: 29496

Tree Height: 22
```

在这个案例中，我们可以从块的平均大小和最大块的大小来了解内存块的大小分布情况。如果需要将一个较大的活对象移到 Tenured 区，但我们已经用完了能够支持该对象的足够大的块，那么垃圾收集晋升就退化成了这种并发模式失败。

为了压缩堆和合并空闲空间链表，JVM 将回退到 Parallel GC，这可能会引发一次时间较长的 STW 暂停。这种分析在实时执行时很有用，因为它会发出信号来提醒我们很快会有一次时间较长的暂停。我们可以通过解析日志来监测该信号，或者使用 Censum 这样的工具来自动检测即将到来的 CMF。

8.6 调优G1

调优 G1 的总体目标是让终端用户只需设置堆的最大值和 `MaxGCPauseMillis`，其他所有事情都由收集器来处理。然而，目前的实际情况离这个目标还有一定的距离。

就像 CMS 一样，G1 自带了大量的配置选项，其中一些还处于实验阶段，效果如何还不能完全看出来。这使得我们很难理解任何调优变化带来的影响。如果调优需要这些选项（目前它们适用于一些调优场景），则必须指定以下这个开关：

```
-XX:+UnlockExperimentalVMOptions
```

特别是，如果要使用选项 `-XX:G1NewSizePercent=<n>` 或 `-XX:G1MaxNewSizePercent=<n>`，则必须指定这个选项。在未来的某个时间，其中的一些选项可能会成为主流，不再需要实验性选项标志，但目前还没有路线图。

图 8-7 显示了一个有趣的 G1 堆视图，该图是用 regions 这个应用程序绘制的。

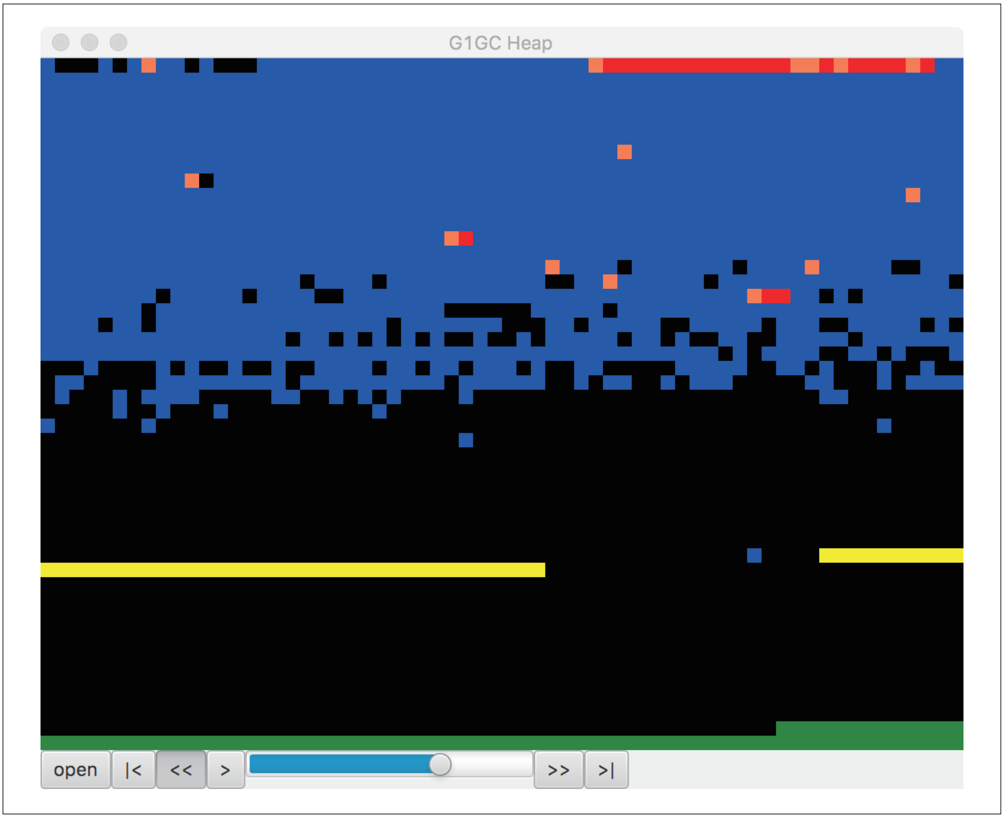


图 8-7：在 regions 中以可视化方式显示 G1 堆的情况

regions 是一个很小的开源 Java FX 应用程序，它可以解析 G1 的垃圾收集日志，还可以根据日志以可视化方式显示 G1 堆的区域布局随时间的变化情况。该工具由 Kirk Pepperdine 编写，可以从 GitHub 上获得。在本书编写时，它仍处于活跃的开发之中。

G1 调优的一个主要问题是，与早期设计相比，其内部结构发生了很大的变化。这就导致出现了“按照坊间传说调优”这样的大问题，因为关于 G1 的很多早期文章，效果已经很有限了。

随着 G1 从 Java 9 开始成为默认的收集器，性能工程师们肯定会被迫解决如何调优 G1 这个问题，但在撰写本书时，这仍然是件苦差事，因为最佳实践仍在不断涌现。

本节最后让我们把目光集中在已经取得的一些进展以及 G1 确实会超越 CMS 这一事实上。回想一下，因为 CMS 并不执行压缩，所以随着时间的推移，堆会碎片化。这最终会导致并发模式失败，并且 JVM 需要执行一次完整的并行收集（可能会出现一次较长的 STW 暂停）。

在 G1 的情况下，只要收集器能跟上分配率，那么增量压缩就可能完全避免并发模式失败。因此对于分配率高且稳定，而且大部分对象寿命很短的应用程序，我们应该：

- 将新生代设置得较大；
- 增加晋升阈值，可能达到最大值（15）；
- 设置该应用程序可以忍受的最长暂停时间。

以这种方式配置 Eden 和 Survivor 空间，可以尽可能使真正的短寿命对象不被晋升。这既减轻了老年代的压力，又减少了清理旧区域的需要。在垃圾收集调优中，虽然并不能保证能达到这种效果，但是在这个负载的例子中，G1 可能确实明显好于 CMS，尽管我们需要在堆的调优上付出一些努力。

8.7 jHiccup

我们已经在 5.3.2 节见过了 HdrHistogram。jHiccup 是一个与它有关的工具，可以从 GitHub 上找到。这是一个注入工具，专门用来显示 JVM 无法连续运行的“间隔”（hiccup）。产生间隔的一个非常常见的原因是垃圾收集 STW 暂停，但其他与操作系统和平台相关的影响也会导致它。因此，它不仅适用于垃圾收集调优，还适用于超低延迟工作。

事实上，我们已经看到过 jHiccup 如何工作的例子。我们在 7.5 节介绍了 Shenandoah 收集器，并展示了该收集器与其他收集器的性能对比图，图 7-9 就是由 JHiccup 生成的。

> 在调优 HotSpot 时，jHiccup 是很好的工具，尽管原作者（Gil Tene）坦承，写它就是为了显示与 Azul 的 Zing JVM 相比，HotSpot 存在的一个缺点。

jHiccup 一般通过 `-javaagent:jHiccup.jar` Java 命令行开关，以 Java 代理的形式使用。它也可以通过 Attach API 来使用（像其他一些命令行工具一样），形式如下：

```
jHiccup -p <pid>
```

这实际上将 jHiccup 注入到了一个正在运行的应用程序中。

jHiccup 以直方图日志的形式产生输出，这些日志可以被 HdrHistogram 读取。首先回顾一下 6.6 节介绍的应用程序分配模式来看看这个操作。

下面我们使用一组适当的垃圾收集日志，并让 jHiccup 以代理方式运行，来看一小段进行运行设置的 shell 脚本：

```
#!/bin/bash

# Simple script for running jHiccup against a run of the model toy allocator

CP=./target/optimizing-java-1.0.0-SNAPSHOT.jar

JHICCUP_OPTS=
  -javaagent:~/.m2/repository/org/jhiccup/jHiccup/2.0.7/jHiccup-2.0.7.jar

GC_LOG_OPTS="-Xloggc:gc-jHiccup.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps
  -XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution"
```

```
MEM_OPTS="-Xmx1G"

JAVA_BIN=`which java`

if [ $JAVA_HOME ]; then
    JAVA_CMD=$JAVA_HOME/bin/java
elif [ $JAVA_BIN ]; then
    JAVA_CMD=$JAVA_BIN
else
    echo "For this command to run, either $JAVA_HOME must be set, or java must be
    in the path."
    exit 1
fi

exec $JAVA_CMD -cp $CP $JHICCUP_OPTS $GC_LOG_OPTS $MEM_OPTS
  optjava.ModelAllocator
```

这将产生一个垃圾收集日志和一个 .hlog 文件。jHiccup 提供了一个可以处理该文件的名为 `jHiccupLogProcessor` 的工具。图 8-8 演示的是一个简单的、可以直接获取的 jHiccup 视图。

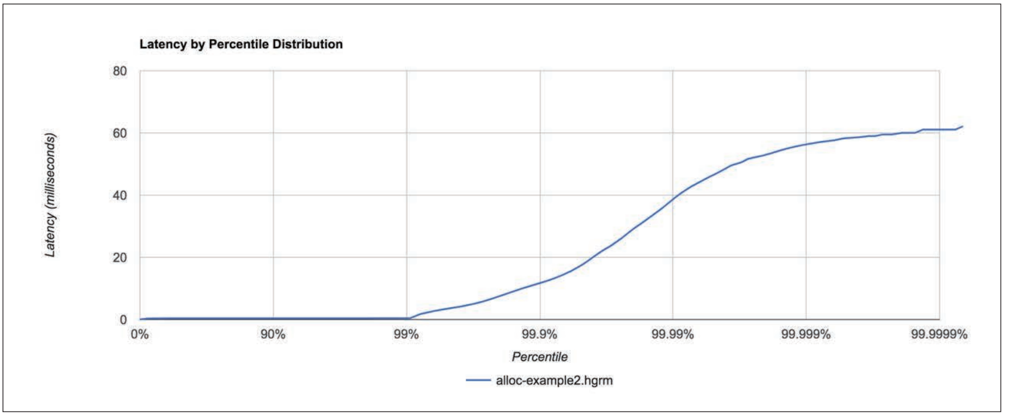


图 8-8：ModelAllocator 的 jHiccup 视图

这是由一个非常简单的 jHiccup 调用生成的：

```
jHiccupLogProcessor -i hiccup-example2.hlog -o alloc-example2
```

这个工具还有其他一些有用的开关，可以使用如下命令来查看所有可用选项：

```
jHiccupLogProcessor -h
```

通常情况下，性能工程师需要理解同一应用程序行为的多个视图，所以为了完整起见，来看看使用 Censum 运行相同应用的结果，如图 8-9 所示。

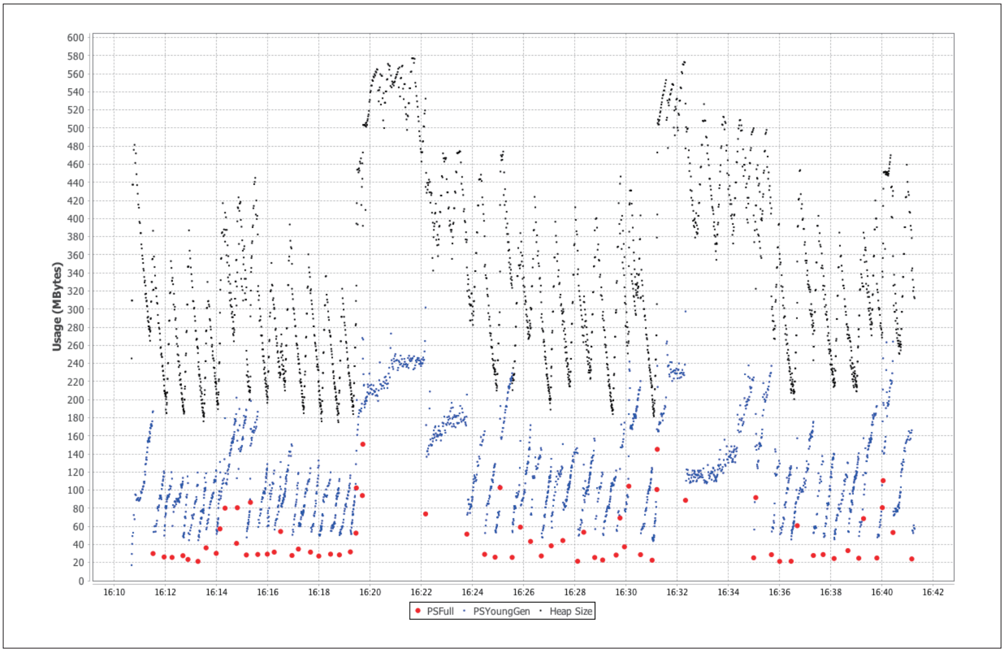


图 8-9：ModelAllocator 的 Censum 视图

从垃圾收集后堆大小的图可以看出，HotSpot 试图调整堆大小，但无法找到一个稳定的状态。这是一种常见的情况，即使对于像 ModelAllocator 这样简单的应用程序也是如此。JVM 是一个非常动态的环境，大多数情况下开发人员应该避免过度关注垃圾收集自动优化的底层细节。

最后，关于 HdrHistogram 和 jHiccup 的一些非常有趣的技术细节，可参阅由 Nitsan Wakart 所写的一篇很棒的博客文章。

8.8　小结

本章介绍了垃圾收集调优艺术的一些皮毛。这里演示的技术大多是针对个别收集器的，但也有一些普遍适用的基本技术。本章还介绍了一些处理垃圾收集日志的基本原则以及一些有用的工具。

下一章将讨论 JVM 的另一个主要子系统：应用程序代码的执行。我们首先将概述解释器，然后在此基础上开始讨论 JIT 编译，包括它与标准编译（或者说 AOT 编译）的关系。

第9章

JVM上的代码执行

任何 JVM 都有两个主要服务，一是内存管理，二是为应用程序代码执行提供一个易用的容器。第 6 章至第 8 章深入介绍了垃圾收集，本章将开始讲解代码执行。

> 回忆一下，Java 虚拟机规范（通常简称为 VMSpec）描述了符合规范的 Java 实现需要如何执行代码。

VMSpec 定义了以解释器方式执行 Java 字节码。然而一般说来，与直接执行机器代码的编程环境相比，解释性环境的性能要差些。大部分生产级的现代 Java 环境是通过提供动态编译能力来解决该问题的。

正如第 2 章所讨论的，这种能力称为**即时编译**。它是一种机制，JVM 通过该机制监控正在执行的方法，以确定个别方法是否有资格编译成可直接执行的代码。

本章首先对字节码解释进行简单概述，并说明为什么 HotSpot 不同于你可能熟悉的其他解释器。然后会介绍剖析制导优化（profile-guided optimization，PGO）的基本概念。最后会探讨代码缓存（code cache）以及介绍 HotSpot 编译子系统的基础知识。

下一章将解释 HotSpot 某些最常见优化背后的机制，并说明你应该如何利用这些机制生成快速编译方法，以及它们可以优化到何种程度及其局限性。

9.1　字节码解释概览

2.1 节简单介绍了 JVM 解释器是以栈式机器方式运行的。这意味着和物理 CPU 不同，它没有可用于计算直接保存区域的寄存器，所有要运算的值都被放在**求值栈**（evaluation stack）

上，且栈式机器的指令通过变换栈顶值来工作。

JVM 提供了 3 个保存数据的主要区域：

- **求值栈**，属于特定方法本地；
- 用于临时存储**结果的局部变量**（也属于方法本地）；
- **对象堆**，在方法和线程间共享。

图 9-1 至图 9-5 显示了一系列使用求值栈来执行计算的虚拟机操作。作为一种伪代码形式，Java 程序员应该可以很容易理解这些操作。

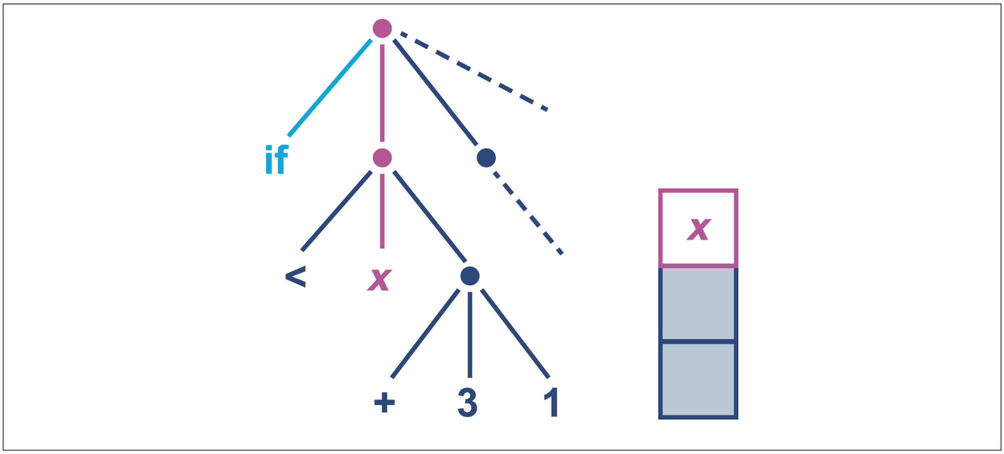


图 9-1：初始解释状态

要确定与 *x* 的内容进行比较的值，解释器现在必须计算右边的子树。

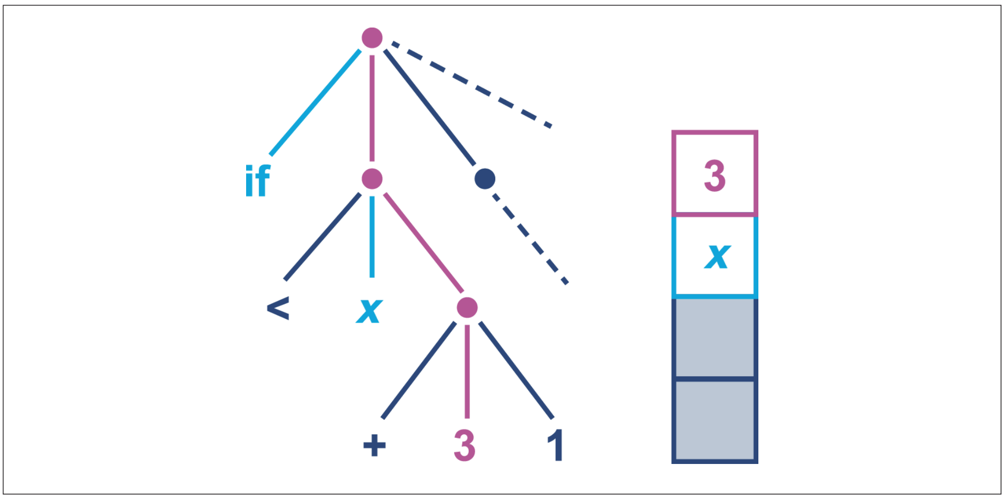


图 9-2：子树求值

下一个子树的第一个值，即一个 `int` 类型的常量 `3`，被加载到栈上。

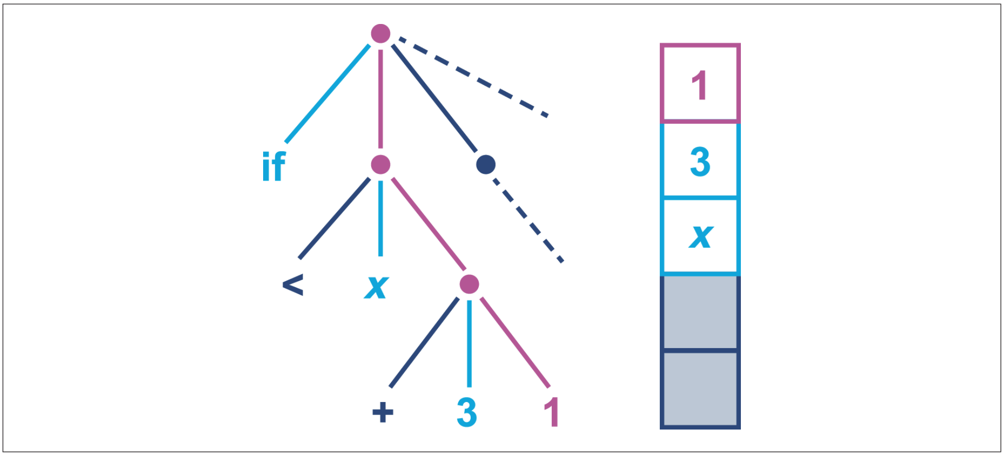


图 9-3：子树求值

现在另一个 `int` 值 `1` 也被加载到栈上。在真正的 JVM 中，这些值将从类文件的常量区域被加载进来。

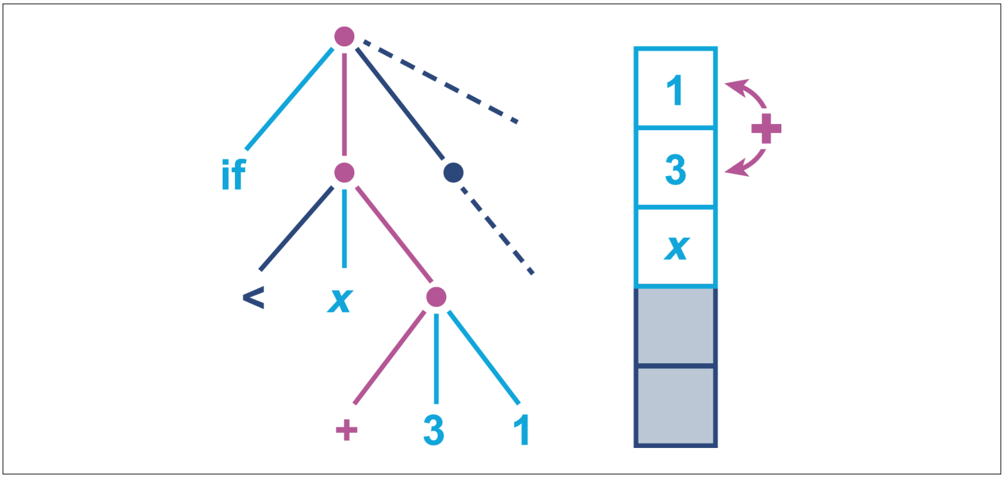


图 9-4：子树求值

这时候加法操作会作用于栈顶的两个元素，你应该移除它们，然后将两数相加的结果放到栈顶。

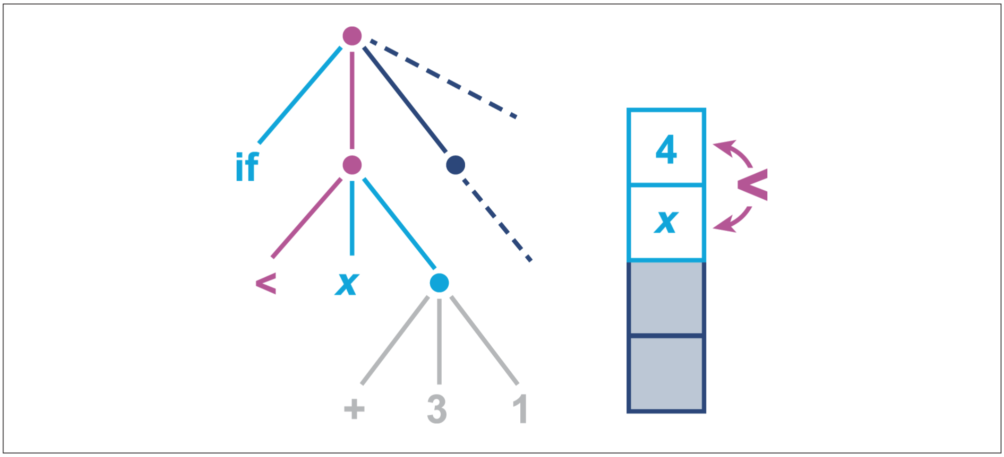


图 9-5：最终的子树求值

生成的值现在可以用于和 *x* 中包含的值进行比较了。在计算其他子树的整个过程中，*x* 的值一直保留在求值栈中。

9.1.1　JVM字节码

在 JVM 中，每个栈式机器操作码（opcode）用 1 字节表示，叫作**字节码**（bytecode）。相应地，操作码的范围是从 0~255，其中大约有 200 个在 Java 10 中使用。

字节码指令是带类型信息的，`iadd` 和 `dadd` 期望在栈顶找到两个正确的基本类型值（分别是两个 `int` 和两个 `double`）。

> 很多字节码指令是以“家族”形式出现的，很多是一条指令用于每个基本类型，一条用于对象引用。

比如，在 `store` 一族中，特定指令有特定意义：`dstore` 的意思是“将栈顶的值保存到一个 `double` 类型的局部变量中”，而 `astore` 的意思是“将栈顶的值保存到一个引用类型的局部变量中”。在这两种情况下，局部变量的类型必须与要存入的值的类型相匹配。

因为 Java 是为高度可移植设计的，所以 JVM 规范被设计为无须修改就能在**大端**（big-endian）和**小端**（little-endian）的硬件架构上运行同样的字节码。因此，JVM 字节码必须决定到底遵循哪种字节顺序（使用相反约定的硬件必须在软件层处理这种差异）。

> 字节码选择了大端方式，所以对于任何多字节序列，高位优先。

一些操作码家族，比如 `load`，有些**简洁形式**。它允许省略参数，从而节省类文件中参数所消耗的字节。特别是，`aload_0` 将当前对象（也就是 `this`）放到栈顶。由于这种操作非常常见，因此可以大大节省类文件的大小。

不过，因为 Java 的类文件通常非常紧凑，所以这种设计决策在 Java 平台发展早期可能更为重要，当时类文件（通常是 Applet）要通过 14.4 kbit/s 的调制解调器下载。

> 从 Java 1.0 开始，只引入一个新的字节码（`invokedynamic`），还有两个（`jsr` 和 `ret`）已经弃用了。

简洁形式和具体类型指令的使用极大地增加了所需要的操作码的数量，这是因为有些是用于表示同样的概念性操作。因此，分配的操作码的数量要远多于字节码所表示的基本操作的数量，而且实际上字节码的概念非常简单。

接下来看一些主要的字节码类别，它们以字节码家族方式排列。注意在下面的表中，`c1` 指示的是一个 2 字节的常量池索引，而 `i1` 指示的是当前方法中的一个局部变量，括号表明该家族有一些简洁形式的操作码。

我们要看到的第一个类别是**加载**和**保存**类，如表 9-1 所示。该类别中的操作码负责将数据移动到栈上，或者从栈上移出来。比如，从常量池中加载数据，或者将栈顶的数据保存到堆中对象的一个字段中。

表9-1：加载和保存类

操作码家族名	参　　数	描　　述
load	(i1)	将局部变量 i1 中的值加载到栈上
store	(i1)	将栈顶的值保存到局部变量 i1 中
ldc	c1	将常量池中 c1 的值加载到栈上
const		将简单常量值加载到栈上
pop		抛弃栈顶的值
dup		复制栈顶的值
getfield	c1	将位于栈顶的对象中的以常量池 c1 位置指示的字段加载到栈上
putfield	c1	将栈顶的值保存到以常量池 c1 位置指示的字段中
getstatic	c1	将以常量池 c1 位置指示的静态字段中的值加载到栈上
putstatic	c1	将栈顶的值保存到以常量池 c1 位置指示的静态字段中

应该弄清楚 `ldc` 和 `const` 的区别。`ldc` 字节码会从当前类的常量池中加载一个常量，其中包括字符串、基本类型常量、类字面值常量以及程序运行所需要的其他（内部）常量。[1]

然而，`const` 操作码不需要参数，它用于加载一系列数量固定的真常量，比如 `aconst_null`、`dconst_0` 和 `iconst_m1`（后者是将 `-1` 当作一个 `int` 值加载）。

注 1：最新的 JVM 版本允许更多外来的常量，以支持现代的高级虚拟机技术。

第二个类别是**算术操作码**，它们仅用于基本类型，并且都不需要参数，因为它们表示的是纯粹的基于栈的操作。这个简单类别如表 9-2 所示。

表9-2：算术类

操作码家族名	描　述
`add`	将栈顶的两个值相加
`sub`	将栈顶的两个值相减
`div`	将栈顶的两个值相除
`mul`	将栈顶的两个值相乘
`(cast)`	将栈顶的值强制转换为一个不同的基本类型
`neg`	对栈顶的值求反
`rem`	计算栈顶的两个值的余数（整数除法）

表 9-3 显示了**流程控制**类操作码。它们用于在字节码层次表示源语言层次的循环和分支等构造。比如，Java 的 `for`、`if`、`while` 和 `switch` 语句都将在源代码编译后被变换为流程控制操作码。

表9-3：流程控制类操作码

操作码家族名	参　数	描　述
`if`	`(i1)`	如果条件为 `true`，就跳转到参数指定位置
`goto`	`i1`	无条件跳转到所给的位移处
`tableswitch`		超出本书范畴
`lookupswitch`		超出本书范畴

关于 `tableswitch` 和 `lookupswitch` 如何工作的详细描述不在本书范畴之内。

虽然流程控制类这个类别看上去不大，但流程控制操作码的实际数量多得惊人。这是因为 `if` 操作码家族的成员有很多，比如在第 2 章 `javap` 的示例中曾看过的 `if_icmpge` 操作码，而且还有很多其他用于表示 Java `if` 语句的不同变种的操作码。

弃用的 `jsr` 和 `ret` 字节码也是这一家族的，不过从 Java 6 开始，`javac` 已经不会输出它们了。对于现代版本的 Java 平台而言，它们已不再合法，所以我们也没有将其放在这个表中。

最重要的操作码类别之一如表 9-4 所示。这就是**方法调用**类别，它是 Java 程序支持将控制转换到一个新方法的唯一机制。换言之，Java 平台把局部的流程控制与将控制转换到另一个方法完全分开了。

表9-4：方法调用类

操作码名称	参　　数	描　　述
`invokevirtual`	`c1`	通过虚拟分派（virtual dispatch）调用在常量池 `c1` 位置找到的方法
`invokespecial`	`c1`	通过“特殊分配”（也就是精确分派）调用在常量池 `c1` 位置找到的方法
`invokeinterface`	`c1,count,0`	使用接口位移查找，调用在常量池 `c1` 位置找到的接口方法
`invokestatic`	`c1`	调用在常量池 `c1` 位置找到的静态方法
`invokedynamic`	`c1,0,0`	动态查找要调用的方法并执行该方法

JVM 的这种设计，以及显式的**方法调用**操作码的使用，意味着在机器代码中没有一个与之等价的 `call` 操作。

JVM 字节码反而使用了一些专业术语。当谈到**调用点**时，它指的是某个方法（调用者）内的一个位置，在那里另一个方法（被调用者）被调用了。不仅如此，在非静态方法调用的情况下，我们总是会将方法解析到某个对象上。这个对象就是所谓的**接收者对象**（receiver object），其运行时类型被称作**接收者类型**（receiver type）。

> 对静态方法的调用总是被变换为 `invokestatic`，且没有接收者对象。

刚接触到虚拟机级别的 Java 程序员可能会惊讶地发现，Java 对象上的方法调用事实上会根据上下文被变换为三种可能的字节码（`invokevirtual`、`invokespecial` 或 `invokeinterface`）之一。

> 我们可以进行这样一个有用的练习：使用 `javap` 反汇编一个简单的 Java 类，编写一些 Java 代码并查看什么情况下会生成什么类型的调用。

实例方法调用通常被转换为 `invokevirtual` 指令，除非我们只知道接收者对象的静态类型是一个接口类型，那时调用会用 `invokeinterface` 操作码来表示。最后，在编译时就知道分派的精确方法，比如 `private` 方法或对父类方法的调用，会生成一个 `invokespecial` 指令。

那么，`invokedynamic` 是怎么加进来的呢？简单来说，就是 Java 在语言级别并没有直接支持 `invokedynamic`，即使在 Java 10 中也没有。

事实上，在 Java 7 中，当 `invokedynamic` 被添加到运行时的时候，根本没有办法强制 `javac` 生成新的字节码。在这个旧版本的 Java 中，只添加了动态调用技术来支持长期实验和非 Java 的动态语言（特别是 JRuby）。

然而，从 Java 8 开始，`invokedynamic` 已经成为 Java 语言的一个重要部分，用于支持一些高级语言特性。下面来看一个 Java 8 Lambda 表达式的简单例子：

```
public class LambdaExample {
    private static final String HELLO = "Hello";

    public static void main(String[] args) throws Exception {
        Runnable r = () -> System.out.println(HELLO);
        Thread t = new Thread(r);
        t.start();
        t.join();
    }
}
```

这个简单的 Lambda 表达式生成的字节码如下：

```
public static void main(java.lang.String[]) throws java.lang.Exception;
  Code:
     0: invokedynamic #2,  0  // InvokeDynamic #0:run:()Ljava/lang/Runnable;
     5: astore_1
     6: new           #3      // class java/lang/Thread
     9: dup
    10: aload_1
    11: invokespecial #4      // Method java/lang/Thread.
                              //          "<init>":(Ljava/lang/Runnable;)V
    14: astore_2
    15: aload_2
    16: invokevirtual #5      // Method java/lang/Thread.start:()V
    19: aload_2
    20: invokevirtual #6      // Method java/lang/Thread.join:()V
    23: return
```

即使对 `invokedynamic` 指令一无所知，从它的形式中也能看出某个方法正被调用，以及调用的返回值会被放到栈顶。

进一步研究这个字节码，我们会发现，不出所料，这个值就是代码中 Lambda 表达式所对应的对象引用，它由正被 `invokedynamic` 指令调用的平台工厂方法创建。当前类的常量池中有些扩展项，这个调用会创建指向扩展项的引用以支持该调用的动态运行时性质。

对 Java 程序员而言，这或许是 `invokedynamic` 最明显的用例了，但它不是唯一的。这个操作码广泛应用于 JVM 上的非 Java 语言中，比如 JRuby 和 Nashorn（JavaScript），在 Java 框架中也是如此。尽管性能工程师应该对此有所了解，不过大多数情况下人们还是因为出于好奇才去了解它的。本书后面部分将介绍与 `invokedynamic` 相关的一些方面。

需要考虑的最后一个类别是**平台操作码**，如表 9-5 所示，它们主要由分配新的堆存储和操纵单个对象上的 intrinsic 锁（同步操作使用的 monitor）等操作组成。

表9-5：平台操作码类别

操作码名称	参　数	描　述
`new`	`c1`	为在常量池的 `c1` 位置找到的类型的对象分配空间
`newarray`	`prim`	为 `prim` 类型的基本类型数组分配空间
`anewarray`	`c1`	为在常量池的 `c1` 位置找到的类型的对象数组分配空间
`arraylength`		获得位于栈顶的数组的长度，并将其置于栈顶

（续）

操作码名称	参　数	描　述
monitorenter		锁定栈顶对象的管程
monitorexit		解锁栈顶对象的管程

对于 newarray 和 anewarray 操作码，执行它们时需要把应该分配的数组长度放到栈顶。

对字节码进行分类时，根据实现每个字节码的复杂程度，“粗粒度”和“细粒度”字节码之间存在明显的区别。

比如，算术操作就是细粒度的，在 HotSpot 中用纯汇编实现。与之相比，粗粒度操作（比如，需要常量池查找，特别是方法分派的操作）则需要调用回到 HotSpot 虚拟机中。

除了单个字节码的语义之外，还应该讨论解释代码中的安全点。第 7 章介绍了 JVM 中安全点的概念，即 JVM 在安全点上需要执行一些管理工作，以及拥有一致的内部状态。这包括对象图，一般来说，它正在被运行的应用程序线程修改。

为达到这种一致性状态，所有应用程序线程都必须停下来，以防它们在 JVM 执行管理的过程中修改共享堆。这是怎么实现的呢？

回想一下，每个 JVM 应用程序线程都是一个真正的操作系统线程。[2] 不仅如此，对于执行解释方法的线程，当一个操作码即将被分派时，应用程序线程肯定在运行 JVM 解释器代码，而不是用户代码。因此堆应该处于一致性状态，而且应用程序线程可以暂停。

所以，“在字节码之间”是暂停应用程序线程的一个理想时间，也是一个最简单的安全点示例。

对于 JIT 编译的方法，情况更为复杂，但本质是必须在 JIT 编译器生成的机器代码中间插入等效的屏障。

9.1.2　简单解释器

正如第 2 章中提到的，我们可以把最简单的解释器想象为套在 while 循环中的一个 switch 语句。Ocelot 项目中有一个这种类型的例子，它是一个为教学设计的 JVM 解释器的部分实现。如果你尚不熟悉解释器的实现，那 Ocelot 0.1.1 就是一个很好的开始。

解释器的 execMethod() 方法负责解释单一的字节码方法。目前只是为支持整型的数学运算和运行“Hello World”程序就已经实现了（有些是假的实现）足够多的操作码。

一个能够处理哪怕非常简单的程序的完整实现，也需要实现非常复杂的操作并使之正常工作，比如常量池查找。然而，即使只有一些非常简单的骨架，解释器的基本结构也是清楚的：

```
public EvalValue execMethod(final byte[] instr) {
    if (instr == null || instr.length == 0)
        return null;
```

注 2：至少主流的服务器 JVM 都是如此。

```
        EvaluationStack eval = new EvaluationStack();

        int current = 0;
        LOOP:
        while (true) {
            byte b = instr[current++];
            Opcode op = table[b & 0xff];
            if (op == null) {
                System.err.println("Unrecognized opcode byte: " + (b & 0xff));
                System.exit(1);
            }
            byte num = op.numParams();
            switch (op) {
                case IADD:
                    eval.iadd();
                    break;
                case ICONST_0:
                    eval.iconst(0);
                    break;
    // ...
                case IRETURN:
                    return eval.pop();
                case ISTORE:
                    istore(instr[current++]);
                    break;
                case ISUB:
                    eval.isub();
                    break;
                // 假实现
                case ALOAD:
                case ALOAD_0:
                case ASTORE:
                case GETSTATIC:
                case INVOKEVIRTUAL:
                case LDC:
                    System.out.print("Executing " + op + " with param bytes: ");
                    for (int i = current; i < current + num; i++) {
                        System.out.print(instr[i] + " ");
                    }
                    current += num;
                    System.out.println();
                    break;
                case RETURN:
                    return null;
                default:
                    System.err.println("Saw " + op + " : can't happen. Exit.");
                    System.exit(1);
            }
        }
    }
```

每次从方法中读取一个字节码，并根据这个字节码进行分派。如果是带参数的操作码，这些参数也会从流中读出来，以确保读取位置总是正确的。

临时值在 `EvaluationStack` 上进行计算，它是 `execMethod()` 中的一个局部变量。算术操作码也在这个栈上进行计算，执行整型的数学计算。

在这个最简单的 Ocelot 版本中没有实现方法调用，但是如果实现了，那么它将继续在常量池中查找某个方法，以找到与要调用的方法相对应的字节码，然后递归调用 `execMethod()`。Ocelot 0.2 演示了这种调用静态方法的情况。

9.1.3 HotSpot特定细节

HotSpot 是一款生产级 JVM，不仅完全实现了各项功能，而且拥有大量的、为支持快速执行而设计的高级特性，从而使得即使在解释模式下也有不错的速度。和我们在 Ocelot 这个用作训练的示例中采用的简单风格不同，HotSpot 是一款**模板解释器**（template interpreter），它会在每次启动时动态构建解释器。

这理解起来要复杂得多，而且对于新手来说，阅读解释器的源代码也是一个挑战。HotSpot 还使用了大量的汇编语言来实现简单的虚拟机操作（如算术运算），并利用原生平台的栈帧布局以进一步提高性能。

同样可能令人惊讶的是，HotSpot 定义并使用了没有出现在虚拟机规范中的、JVM 特有的（也可以说是私有的）字节码。HotSpot 使用这些字节码来区分常用的热点情况和某个特定操作码更一般的使用情况。

这是为了帮助处理数量惊人的边缘情况而设计的。比如，`final` 方法不能被覆盖，所以开发人员可能会认为，当调用该方法时，`javac` 会生成一个 `invokespecial` 操作码。然而，《Java 语言规范》中的 13.4.17 节对这种情况进行了如下说明。

> 将一个声明为 final 的方法修改为不用 final 修饰，不会破坏与现有二进制文件的兼容性。

考虑一段 Java 代码，比如：

```
public class A {
    public final void fMethod() {
        // ……处理
    }
}

public class CallA {
    public void otherMethod(A obj) {
        obj.fMethod();
    }
}
```

现在假设 `javac` 把对 `final` 方法的调用编译成 `invokespecial`。`CallA::otherMethod` 的字节码看起来像这样：

```
public void otherMethod()
  Code:
     0: aload_1
     1: invokespecial #4                  // Method A.fMethod:()V
     4: return
```

接下来假设 A 的代码发生了变化，fMethod() 不再用 final 修饰。现在它可以在子类中被覆盖了，我们将其称为 B。假设 B 的一个实例被传给了 otherMethod()。从字节码看，invokespecial 指令将被执行，从而导致调用了错误的方法实现。

这破坏了 Java 面向对象的规则。严格来说，它违反了**里氏替换原则**（以面向对象编程的先驱者之一 Barbara Liskov 的名字命名）。简单来说，这个原则就是子类的实例可以在任何需要超类的实例的地方使用，它也是著名的软件工程 SOLID 原则中的 L。

因此，对 final 方法的调用必须编译成 invokevirtual 指令。然而，由于 JVM 知道这样的方法不能被覆盖，因此 HotSpot 解释器会有一个私有的字节码，专门用于分派 final 方法。

再举一个例子。语言规范中说，如果一个对象要使用终结机制（有关终结机制的讨论，参见 11.6 节），则必须到终结子系统中注册，而且该注册必须在超类 Object 的构造器 Object::<init> 调用完成后立即进行。因为在 JVMTI 和其他有可能重写字节码的情况下，这个代码位置可能就不是那么清楚了。为了确保严格的一致性，HotSpot 有一个私有的字节码，用来标记从真正的 Object 构造器的返回。

9.2 AOT编译和JIT编译

本节将讨论和比较 AOT 编译和 JIT 编译这两种生成可执行代码的替代方案。

相对而言，AOT 编译出现得更早，而 JIT 编译是最近才发展起来的。但在 Java 出现的 20 多年里，这两种方法不但没有停滞不前，反而彼此借鉴了对方很多的成功技术。

9.2.1 AOT编译

如果有过 C 和 C++ 等语言的编程经验，那你一定很熟悉 AOT 编译（你可能只称其为“编译”）。它的操作过程是这样的：一个外部程序（编译器）获取人类可读的程序源码，并直接输出可执行的机器代码。



AOT 编译源代码意味着你只有一次机会来利用任何潜在的优化。

你可能希望生成一个你打算运行在目标平台和处理器架构上的可执行程序，这些与目标平台紧密相关的二进制文件能够利用任何处理器专有的特性来提高程序的运行速度。

然而在大多数情况下，在可执行文件的生成过程中，并没有它所要运行的目标平台的信息。因此，AOT 编译必须保守地选择哪些处理器特性可供使用。如果在编译代码时假设某些特性是可用的，然而在运行时，当前处理器并没有这些特性，那二进制文件就根本无法运行。

这就导致了 AOT 编译的二进制文件通常无法充分利用 CPU 的能力，而潜在的性能提升也只能是纸上谈兵了。

9.2.2 JIT编译

JIT 编译是一种通用的技术，其程序（通常是从某种方便的中间格式）在运行时被转换为高度优化的机器代码。HotSpot 和大多数其他主流的生产级 JVM 在很大程度上依赖这种技术。

这种技术在运行时收集程序的有关信息，并进行性能剖析（profile），该剖析可用于确定程序中哪些部分使用频率较高且能从优化中获益最多。

> 该技术被称为**剖析制导优化**。

由于 JIT 子系统与运行中的程序共享虚拟机资源，因此剖析的成本以及执行任何优化所付出的成本都要和预期的性能收益保持平衡。

由于将字节码编译成原生代码的成本是在运行时消耗的，而所消耗的资源（CPU 周期、内存）原本可以用于应用程序本身的执行，因此需要谨慎执行 JIT 编译。虚拟机会收集与应用程序相关的统计数据（寻找“热点”）以了解最佳的优化位置。

回忆一下图 2-3 所示的整体架构：剖析子系统在记录哪些方法正在运行。如果一个方法的调用次数超过了某个阈值，从而符合了编译条件，那么代码生成子系统就会启动一个编译线程，将字节码转换为机器码。

> 现代版本的 `javac`，其设计就是要生成“愚蠢的字节码”（dumb bytecode）。虽然它所执行的优化非常有限，但提供了一个易于 JIT 编译器理解的程序表示。

我们在 5.1 节介绍了 PGO 导致的 JVM 预热问题。应用程序刚启动时，性能会有一段时间不太稳定，这种现象经常导致 Java 开发人员提出这样的问题：我们不能把编译好的代码保存到磁盘上，在应用程序下次启动时再使用吗？或者每次运行应用程序时都要重新运行优化和编译决策不是很浪费吗？

关键是，所提的这些问题都隐含了一些关于运行应用程序代码的性质的基本假设，而这些假设通常是不正确的。接下来通过一个金融行业的例子来说明这个问题。

美国的失业率数据每月公布一次。发布**非农就业数据**（NFP）的这一天，交易系统中的流量非比寻常，而且在当月的其他时间里通常不会再出现这种情况。如果保存了另外一天的优化，并在 NFP 发布当天运行该优化，则其效果可能不如刚刚计算出来的优化效果好。这将导致使用预先计算的优化的系统实际上没有使用 PGO 的应用程序具有竞争力。

应用程序的性能在不同的应用程序运行之间存在显著差异的行为非常常见。它表明，像 Java 这样的环境应该让开发人员尽量不受此类领域信息的影响。

因此，HotSpot 不会尝试保存任何剖析信息，相反，它会在虚拟机关闭时将其丢弃，所以每次都必须从头开始重新建立剖析信息。

9.2.3 比较AOT和JIT

AOT 编译的优点是相对简单易懂。机器码直接从源代码生成，与编译单元相对应的机器码可以直接以汇编的形式获得，从而让代码具有直接的性能特征成为可能。

当然，有得必有失，AOT 编译也意味着放弃对有价值的运行时信息的访问，而这些信息有助于做出优化决策。现在 `gcc` 和其他编译器中也开始应用诸如链接时优化（link-time optimization，LTO）和某种形式的 PGO 等技术，不过与 HotSpot 中的相关技术相比，它们还处于早期开发阶段。

在 AOT 编译过程中，针对处理器的特定特性，将产生一个只与该处理器兼容的可执行文件。这种技术对于要求低延迟或极端性能的应用程序很有用。如果构建应用程序所用的硬件和它的运行环境完全相同，那么编译器可以利用所有可用的处理器进行优化。

然而，这种技术并不能扩展，如果想在一系列目标架构上实现性能最大化，则需要为每个架构生成一个单独的可执行文件。

相比之下，HotSpot 可以在新处理器特性发布时为它们添加优化，而应用程序不必重新编译其类文件和 JAR 包来利用这些特性。随着 JIT 编译器的改进，很容易发现在 HotSpot 虚拟机的各个新版本中，程序性能在显著提高。

接下来再来讨论“Java 程序不能被 AOT 编译”这个流传已久的谬论。事实上，提供对 Java 程序进行 AOT 编译功能的商用虚拟机已经出现好几年了，而且在某些环境中，AOT 还是部署 Java 应用程序的主要途径。

最后，从 Java 9 开始，HotSpot 虚拟机已经开始提供 AOT 编译功能了，最初针对的是核心 JDK 类。这是向着从 Java 源代码生成 AOT 编译的二进制文件迈出的、相当有限的第一步，不过这也代表着 Java 开始告别过去普遍使用的传统 JIT 环境。

9.3 HotSpot JIT基础

因为 HotSpot 中的基本编译单元是一个完整的方法，所以一个方法所对应的所有字节码都会被一次性地编译为原生代码。HotSpot 还支持使用一种叫作**栈上替换**（on-stack replacement，OSR）的技术来支持热循环的编译。

OSR 用来帮助处理这样的情况：一个方法的调用频率还没有多到可以触发编译，但是该方法中包含了一个循环，如果此循环体本身就是一个方法，那么它就可以被编译。

HotSpot 使用 `klass` 元数据结构中存在的虚函数表（由对象的 `oop` 中的 `klass` 字指向）作为实现 JIT 编译的主要机制，我们将在下一节中看到。

9.3.1 Klass字、虚函数表和指针变换

HotSpot 是一个多线程的 C++ 应用程序。这么说似乎过于简单，但需要记住的是，从操作系统的角度来看，每一个正在执行的 Java 程序实际上都是某个多线程应用程序的一部分。即使是单线程应用程序，也总和多个虚拟机线程一起执行。

HotSpot 中最重要的线程组之一就是组成 JIT 编译子系统的线程，其中包括探测方法何时达到编译条件的剖析线程，以及负责生成实际机器码的编译器线程本身。

总体情况是，当指示进行编译时，该方法会被放置在一个编译器线程上并在后台进行编译，整个过程如图 9-6 所示。

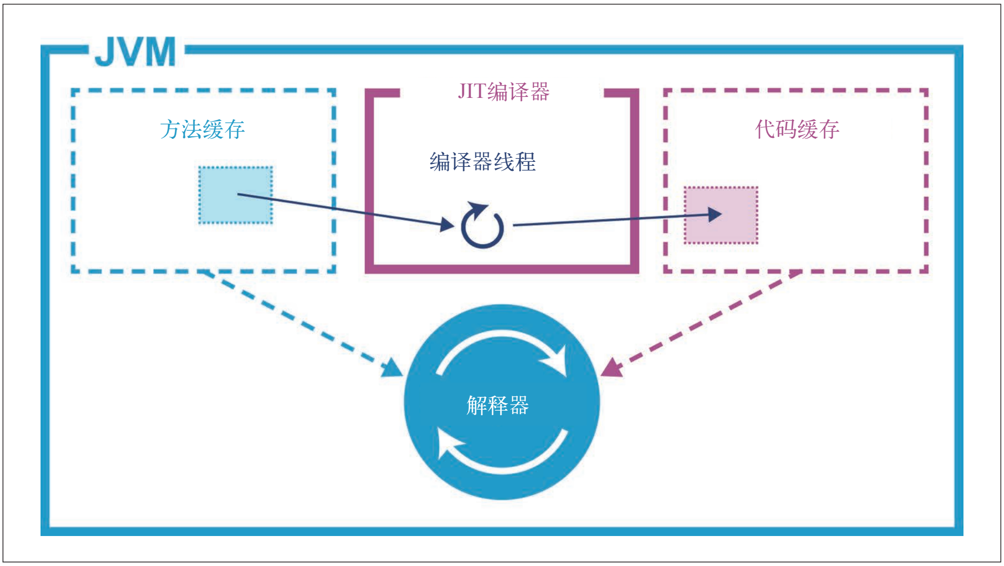


图 9-6：单个方法的简单编译

在得到优化的机器代码后，相关 klass 的虚函数表（vtable）中的条目会被更新，以指向新编译的代码。



vtable 的指针更新这个操作有个略显奇怪的名字——**指针变换**（pointer swizzling）。

这意味着对该方法的任何新调用都会得到编译版本的代码，而当前正在执行解释版本代码的线程会在解释模式下完成当前调用，但在下一次调用时会使用新的编译版本代码。

OpenJDK 已经被广泛移植到很多不同的架构上，其中 x86、x86-64 和 ARM 是其主要的目标平台，同时对 SPARC、Power、MIPS 和 S390 也有不同程度的支持。Oracle 官方支持 Linux、macOS 和 Windows 这 3 个平台作为操作系统。有些开源项目本身支持更多的操作系统，包括 BSD 和嵌入式系统。

9.3.2 JIT编译日志

一个所有性能工程师都应该知道的重要 JVM 开关是：

```
-XX:+PrintCompilation
```

这将导致在 STDOUT（标准输出）上生成一个编译事件的日志，从而使工程师可以基本了解正在编译的内容。

例如，如果这样调用示例 3-1 中的缓存的例子：

```
java -xx:+PrintCompilation optjava.Caching 2>/dev/null
```

那么产生的日志（在 Java 8 下）看起来将是这样的：

```
56   1       3  java.lang.Object::<init> (1 bytes)
57   2       3  java.lang.String::hashCode (55 bytes)
58   3       3  java.lang.Math::min (11 bytes)
59   4       3  java.lang.String::charAt (29 bytes)
60   5       3  java.lang.String::length (6 bytes)
60   6       3  java.lang.String::indexOf (70 bytes)
60   7       3  java.lang.AbstractStringBuilder::ensureCapacityInternal (27 bytes)
60   8     n 0  java.lang.System::arraycopy (native)   (static)
60   9       1  java.lang.Object::<init> (1 bytes)
60   1       3  java.lang.Object::<init> (1 bytes)   made not entrant
61  10       3  java.lang.String::equals (81 bytes)
66  11       3  java.lang.AbstractStringBuilder::append (50 bytes)
67  12       3  java.lang.String::getChars (62 bytes)
68  13       3  java.lang.String::<init> (82 bytes)
74  14 %     3  optjava.Caching::touchEveryLine @ 2 (28 bytes)
74  15       3  optjava.Caching::touchEveryLine (28 bytes)
75  16 %     4  optjava.Caching::touchEveryLine @ 2 (28 bytes)
76  17 %     3  optjava.Caching::touchEveryItem @ 2 (28 bytes)
76  14 %     3  optjava.Caching::touchEveryLine @ -2 (28 bytes)   made not entrant
```

请注意，由于绝大多数的 JRE 标准库是用 Java 编写的，因此它们可以与应用程序代码一起进行 JIT 编译。所以如果你在编译后的代码中看到很多非应用方法，不要感到惊讶。

即使是在一个非常简单的基准测试上，每次运行所得到的被编译方法的确切集合可能也都略有不同。不用担心，这是 PGO 的动态特性的一个副作用。

PrintCompilation 的输出会以相对简单的方式格式化。首先是一个方法被编译的时间（自虚拟机启动以来以毫秒为单位），接下来是一个数字，表示该方法在这次运行中被编译的顺序。其他的一些字段如下。

- n：方法是原生的
- s：方法是同步的
- !：方法有异常处理程序
- %：方法是通过栈上替换编译的

PrintCompilation 提供的细节有限。要访问更多关于 HotSpot JIT 编译器所做决策的详细信息，可以使用：

```
-XX:+LogCompilation
```

这是一个诊断选项，必须使用一个附加标志来解锁：

```
-XX:+UnlockDiagnosticVMOptions
```

这指示虚拟机输出一个包含 XML 标记的日志文件，该标记表示从字节码向原生代码转换过程中与队列和优化相关的信息。`LogCompilation` 标志会很冗长，并生成数百 MB 的 XML 输出。

然而，正如下一章将介绍的开源的 JITWatch 工具，它可以解析这个文件，并以更容易理解的格式呈现这些信息。

其他的虚拟机，比如提供了 Testarossa JIT 的 IBM J9，也可以将 JIT 编译器信息记录到日志中，不过 JIT 日志并没有标准的格式，因此开发人员必须学会解释每种日志格式，或学着使用适当的工具。

9.3.3　HotSpot中的编译器

HotSpot JVM 中实际上有两种 JIT 编译器，而不是一种。它们的正式名称分别为 C1 和 C2，但有时也被称为客户端编译器和服务器编译器。过去，C1 用于 GUI 应用程序和其他“客户端”程序，而 C2 用于长期运行的“服务器”应用程序。现代的 Java 应用程序通常模糊了这一区别，而 HotSpot 利用了这一新趋势。

> 一个编译后的代码单元被称为 nmethod（national method 的缩写）。

两种编译器所采取的一般方法都是依赖一个关键的测量值来触发编译：一个方法被调用的次数，或者是**调用计数**（invocation count）。一旦这个计数器达到某个特定阈值，虚拟机就会收到通知，并考虑将该方法加入编译队列。

编译过程首先会为方法创建一个内部表示，接下来将结合在解释阶段所收集的剖析信息进行优化。然而，C1 和 C2 所产生的代码的内部表示非常不同。与 C2 相比，C1 被设计得更简单，编译时间也更短。权衡的结果是，C1 没有像 C2 那样得到充分的优化。

两者共同使用的一种技术是**静态单一赋值**（static single assignment，SSA）。这本质上是将程序转换为一种这样的形式：每个变量只被赋值一次，不会发生重新赋值。用 Java 的编程术语来说就是程序实际上被重写为只包含 `final` 变量。

9.3.4　HotSpot中的分层编译

自 Java 6 以后，JVM 支持一种称为**分层编译**（tiered compilation）的模式，其大致内容通常就是，在解释模式下运行，直到触发了简单的 C1 编译形式，当 C2 完成了更高级的优化时，再切换到使用该编译的代码。

然而，这种描述并不完全准确。可以从源文件 advancedThresholdPolicy.hpp 中看到虚拟机内有 5 个可能的执行层次。

- 第 0 层：解释器
- 第 1 层：C1，开启全部优化，不开启剖析
- 第 2 层：C1，支持调用和回边计数器
- 第 3 层：C1，开启全部剖析功能
- 第 4 层：C2

表 9-6 显示出每种编译路径并不会利用到每个层次。

表9-6：编译路径

路　径	描　述
0-3-4	解释器，开启全部剖析功能的 C1，C2
0-2-3-4	解释器，在 C2 忙碌的情况下，快速用 C1 编译，然后是完全编译的 C1，然后是 C2
0-3-1	简单方法
0-4	无分层编译（直接到 C2）

在简单方法的情况下，该方法一开始会像往常一样被解释，但之后在 C1 开启全部剖析功能时就能确定该方法是简单方法了。这意味着 C2 编译器显然不会产生更好的代码了，因此编译终止。

分层编译成为默认设置已经有一段时间了，在性能调优过程中通常不需要调整其操作。但是知道其操作是有用的，因为它经常会使所观察到的编译方法的行为复杂化，并可能误导粗心的工程师。

9.4 代码缓存

JIT 编译的代码被存储在一个叫作**代码缓存**（code cache）的内存区域。该区域还存储了属于虚拟机本身的其他原生代码，比如解释器的部分内容。

代码缓存在虚拟机启动时设置了一个固定的最大值。它不能超出这个限制，所以有可能被填满。如果被填满，就不能再进行 JIT 编译，并且未编译的代码只能在解释器中执行。这将对应用程序的性能产生影响，并可能导致它明显低于潜在的最大性能。

代码缓存被实现为一个堆，其中包含一个未分配区域和一个空闲链表（指向被释放的块）。当原生代码被删除时，该代码对应的块就会被添加到空闲链表中。会有一个叫作**清扫器**（sweeper）的进程来负责回收这些块。

当要存储一个新的原生方法时，可以在空闲链表中搜索一个足够大的块来存储编译后的代码。如果没有找到，那么只要代码缓存有足够的空闲空间，就会从未分配的空间中创建一个新块。

在以下情况中，原生代码可以从代码缓存中删除：

- 它被取消了优化（基于某种假设进行了推测性优化，结果证明条件不成立）；
- 它被替换为另一个编译版本（在分层编译的情况下）；
- 包含该方法的类被卸载了。

可以使用以下这个虚拟机开关来控制代码缓存的最大值：

```
-XX:ReservedCodeCacheSize=<n>
```

注意，在启用了分层编译之后，因为 C1 客户端编译器的编译阈值更低，所以将有更多方法达到阈值。因此，会设置更大的默认值来容纳这些额外的编译方法。

在 Linux x86-64 上的 Java 8 中，代码缓存默认的最大值为：

```
251658240 (240MB) when tiered compilation is enabled (-XX:+TieredCompilation)
 50331648  (48MB) when tiered compilation is disabled (-XX:-TieredCompilation)
```

碎片

在 Java 8 和更早的版本中，如果 C1 编译器生成的很多中间编译结果在被 C2 编译的结果取代之后被移除，那么在代码缓存中就会产生很多碎片。这可能会导致未分配的区域被耗尽，并且使所有的空闲空间都在空闲链表中。

此时，代码缓存分配器将不得不遍历链表，直到找到一个足够大的块来容纳新编译的原生代码。反过来，清扫器也必须做更多的工作来扫描以寻找可以回收到空闲链表中的块。

最后，不管是哪种垃圾收集机制，如果不对内存块进行迁移，都会受到碎片的影响，代码缓存也不例外。

如果没有压缩（compact）整理方案，代码缓存就会出现碎片化，进而可能导致编译停止，毕竟这只是缓存耗尽的另一种形式。

9.5　简单JIT调优

在进行代码调优练习时，要确保应用程序确实利用了 JIT 编译的优势相对来说是比较容易的。

简单的 JIT 编译调优的一般原则很简单：任何想要编译的方法，都应该拿到足够的资源来操作。为了达到这个目的，请按照下面这个简单的检查清单来检查。

1. 首先打开 `PrintCompilation` 开关并运行应用程序。
2. 收集显示哪些方法被编译的日志。
3. 现在通过 `ReservedCodeCacheSize` 来增加代码缓存的大小。
4. 重新运行应用程序。
5. 查看缓存增大后已编译方法的集合。

性能工程师需要考虑 JIT 编译所固有的、轻微的不确定性。牢记这一点，我们很容易就能观察到一些明显的迹象：

- 当缓存大小增加时，已编译方法的集合是否以某种有意义的方式变大了？
- 是否所有对主事务路径非常重要的方法都被编译了？

如果已编译方法的数量并没有随着缓存大小的增加而增加（说明代码缓存没有被充分利用），那么只要负载模式具有代表性，JIT 编译器就不会缺乏资源。

此时应该可以直接确认所有属于事务热路径上的方法都出现在编译日志中了。如果没有，那么下一步就是确定根本原因——为什么这些方法没有被编译。

实际上，这个策略就是通过保证 JVM 不会耗尽代码缓存空间，从而确保 JIT 编译永远不会关闭。

本书最后还将遇到更复杂的技术，尽管 Java 版本之间存在一些细微的差异，但简单的 JIT 调优方法可以帮助提升大量应用程序的性能。

9.6 小结

JVM 的初始代码执行环境是字节码解释器。本章探讨了解释器的基础知识，因为要正确理解 JVM 的代码执行，掌握字节码如何工作的知识是必不可少的。此外，我们还介绍了 JIT 编译的基本理论。

然而，对于大多数性能工作而言，JIT 编译的代码的行为远比解释器的任何方面重要。下一章将在本章介绍的基础上，深入研究 JIT 编译的理论与实践。

对于许多应用程序而言，本章所演示的针对代码缓存的简单调优技术已经足够了。但对于那些对性能特别敏感的应用程序来说，可能还需要对 JIT 行为进行更深入地探索。下一章将介绍一些对要求更严格的应用程序进行调优的工具和技术。

第 10 章

理解即时编译

本章将深入介绍 JVM 中 JIT 编译器的内部工作方式。大部分内容直接适用于 HotSpot，不过并不保证和其他 JVM 实现一致。

我们曾经提到过，与 JIT 编译相关的科学研究已经相当深入，不只是 JVM，很多现代编程环境也用到了 JIT。因此，很多 JIT 技术也适用于其他 JIT 编译器。

因为这个主题非常抽象，而且技术上也很复杂，所以需要一些工具来帮我们理解 JVM 的内部工作方式，并将其以可视化方式表现出来。本章将使用的主要工具是 JITWatch，我们会先加以介绍。之后会解释具体的 JIT 优化和特性，并演示如何通过 JITWatch 观察该技术及其效果。

10.1　认识JITWatch

JITWatch 是一款开源的 JavaFX 工具，由本书的作者之一（克里斯·纽兰）作为个人项目设计和构建。该工具现在被托管在 AdoptOpenJDK 倡议之下，作为伦敦 Java 社区（London Java Community）为提高 Java 生态系统的社区参与度而运行的程序的一部分。

JITWatch 能够让开发团队或 DevOps 团队更好地理解 HotSpot 在应用程序执行期间实际对字节码做了什么。我们可以通过调整编译开关来提高应用程序的性能，但是必须有一个机制来衡量我们所做的任何改进。

> 任何要分析的方法都必须在热路径（hot path）中使用，并且有资格进行编译。被解释的方法不适合作为严肃优化的目标。

JITWatch 为比较提供了客观的测量结果。如果没有这些测量结果，我们就可能陷入 4.4 节讨论的“忽略大局”反模式。

JITWatch 的工作方式是，解析和分析正在运行的 Java 应用程序所输出的 HotSpot 详细编译日志，并在 JavaFX 图形用户界面中显示出来。这意味着应用程序在运行的时候需要打开一组特定的标志。

除了应用程序正常运行所需的标志之外，如果没有以下标志，则必须为 JITWatch 添加以下标志：

```
-XX:+UnlockDiagnosticVMOptions -XX:+TraceClassLoading -XX:+LogCompilation
```

在打开这些标志后，JVM 将生成一个供 JITWatch 使用的日志。

10.1.1　基本的JITWatch视图

在应用程序运行结束后，你可以启动 JITWatch 和加载日志，以及基于应用程序的实际运行生成一个如图 10-1 所示的视图。

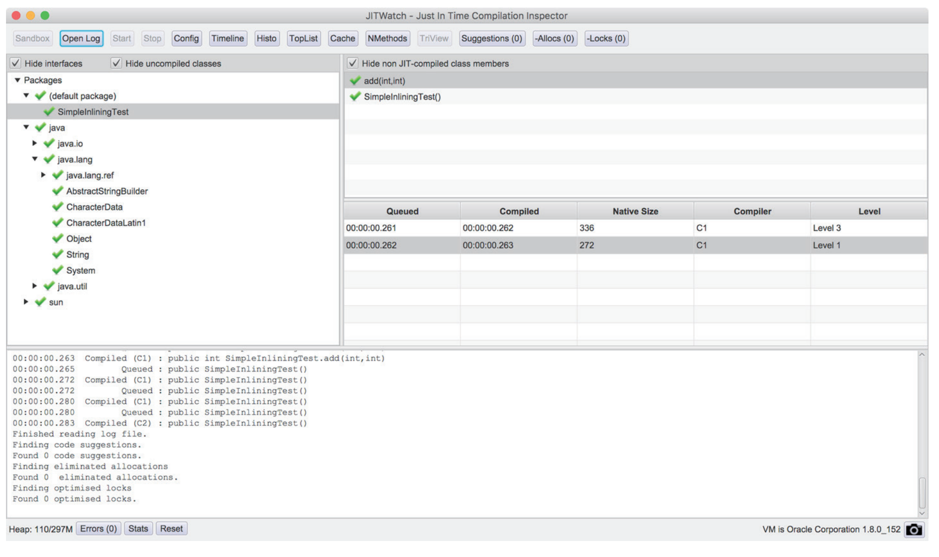


图 10-1：JITWatch 主窗口

除了加载执行完毕的程序的日志，JITWatch 还提供一个叫作**沙箱**（sandbox）的环境，用于实验 JIT 行为。可以借助这个视图为小型程序快速建立原型，看看 JVM 所做出的 JIT 决策。图 10-2 是一个示例视图。

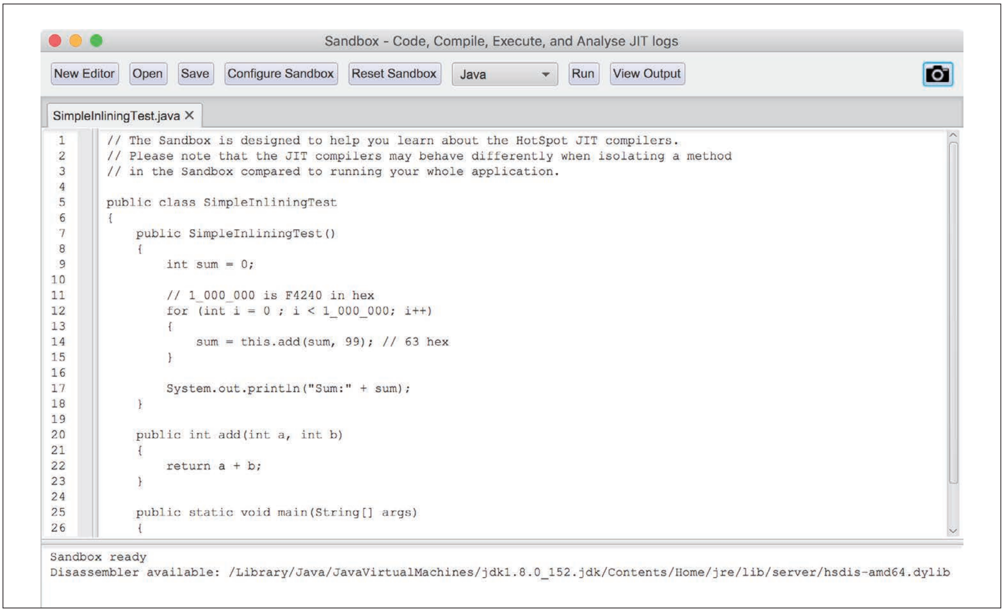


图 10-2：JITWatch 沙箱

作为加载已有日志文件的替代方案，沙箱工作流程支持创建或加载一个用 Java 或其他支持的 JVM 语言编写的程序。

点击运行（Run）按钮，JITWatch 会执行以下操作。

1. 把你的程序编译为字节码。
2. 在 JVM 上执行该程序，打开 JIT 日志。
3. 把 JIT 日志文件加载到 JITWatch 中进行分析。

有些小的改变可能会影响 JVM 所做的优化选择，沙箱就是为了对这些改变进行快速反馈而设计的。除了 Java 之外，你还可以通过配置 Scala、Kotlin、Groovy 和 JavaScript（Nashorn）的路径来使用沙箱。

沙箱非常有用，所以你应该注意编辑窗口中显示的警告。请记住，在沙箱中运行的代码，其表现可能与真实应用有很大不同。

沙箱还支持对控制 JIT 子系统的虚拟机开关进行实验（参见图 10-3）。例如，通过修改沙箱的配置可以改变 JVM 的 JIT 行为，其中包括：

- 输出反汇编原生方法（需要在 JRE 中安装一个反汇编二进制工具，如 hsdis），还可以选择汇编语法；
- 覆盖 JVM 默认设置的分层编译（使用 C1 和 C2 JIT 编译器）；
- 禁用压缩指针（这样可以去掉地址移位指令，让汇编代码更易读）；

- 禁用栈上替换；
- 覆盖内联的默认限制。

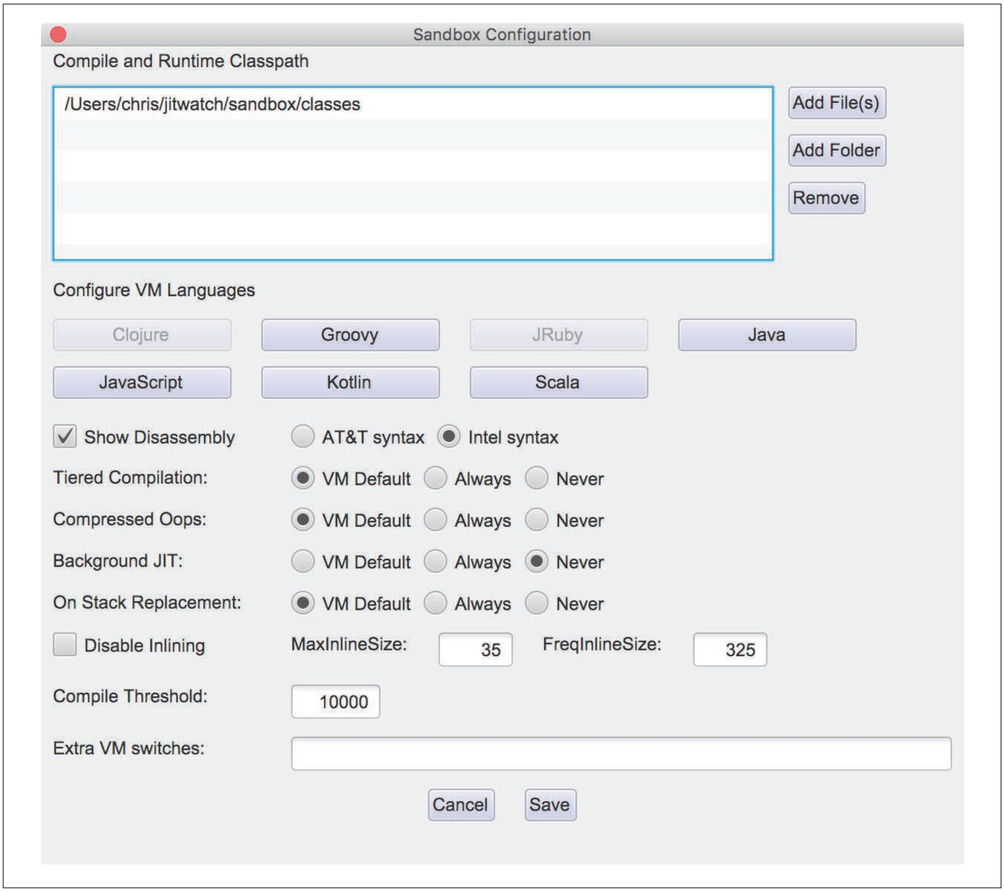


图 10-3：JITWatch 沙箱配置

修改这些设置会对 JVM 的性能产生很大的影响。除了极端情况之外，不建议在生产系统中修改它们。即使是在极端情况下，如果没有进行充分的测试，也不应该修改它们。

沙箱与正常运行的 JVM 应用程序还有一个区别，即当运行完整的应用程序时，JVM 能够在更多的代码上组合使用各种优化，而在沙箱中 JVM 只能看到一些片段。激进的 JIT 编译器（如 C2）甚至可以首先应用一组优化，使优化器能够看到更大范围的程序代码。

比如，内联技术会将方法合并到调用位置。这意味着优化器现在可以考虑进一步的优化（比如进行更多内联或其他形式）了，而这些优化在内联之前并不明显。反过来，这也意味着 JIT 编译器会以不同方式对待沙箱中只有无关紧要的内联行为的玩具应用程序，以及具有真正内联的真实应用中的方法。

因此，与沙箱相比，大多数从业者更希望有一个可以对应用程序编译了解更深入的视图。幸运的是，JITWatch 的主视图功能非常强大，叫作 TriView。这个视图会显示源代码是如何被编译成字节码和汇编代码的。示例如图 10-4 所示，它可以证明 JIT 编译器删除了一些不必要的对象分配——这是现代 JVM 中的一项重要优化，本章后面会介绍。

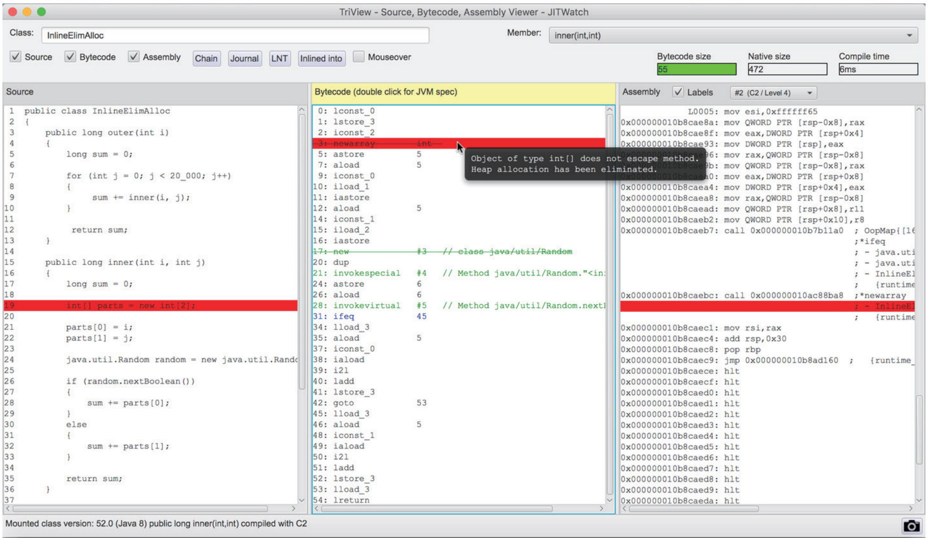


图 10-4：JITWatch TriView

JITWatch 支持可视化地查看每个编译后的方法在代码缓存中的存储位置。这是一个比较新的功能，还在积极开发中，目前的视图如图 10-5 所示。

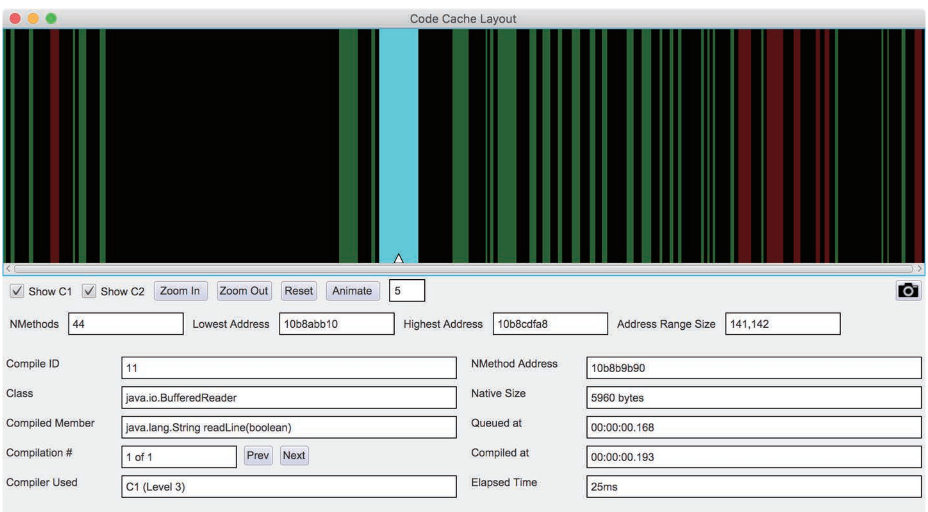


图 10-5：JITWatch 代码缓存布局

在 Java 8 和更早的版本中，代码缓存会将剖析过的编译方法、未剖析的编译方法以及虚拟机自己的原生代码保存在一个单独的区域中。

Java 9 引入了一个分段代码缓存，因此不同类型的原生代码会存储到不同的区域，以减少碎片和清理时间，并改进完全编译代码的局部性。第 15 章在深入讨论 Java 9 的特性时还会探索该特性。

10.1.2 调试JVM和hsdi

如果想更深入地了解 JIT 子系统的调优以及从中获得统计数据，那么你可以用**调试版的 JVM** 来实现这一目的。这种版本是为生成产品版 JVM 所不具备的调试信息而构建的，因此性能上会有所损失。

主要的 JVM 厂商一般不提供调试版虚拟机（Debug VM）的下载方式，但是可以从 OpenJDK 源代码中构建 HotSpot Debug JVM。



Linux x86_64 平台的 Debug VM 二进制文件可以从作者的网站上下载。

如果想检查 JIT 编译器生成的反汇编原生代码，则需要一个像 hsdis 这样的反汇编二进制工具。这可以从 OpenJDK 源代码中构建出来，具体说明见 JITWatch 维基百科上的文章“Building hsdis”。

要让虚拟机输出方法的汇编代码，可以添加如下开关：

```
-XX:+PrintAssembly
```



将原生代码反汇编为可读的汇编语言代码，这是在 JIT 编译器生成该方法之后直接执行的。该操作成本很高，并且会影响程序的性能，所以要小心使用。

介绍完了 JITWatch，下面来看看 HotSpot JIT 编译器的一些技术细节。

10.2 介绍JIT编译

为了补充 JITWatch 中已编译代码的视图，性能工程师需要了解虚拟机如何收集数据，以及它将对正在执行的程序进行哪些优化。

我们已经看到，HotSpot 使用剖析制导优化来指导 JIT 做出编译决策。在后台，HotSpot 将运行程序的剖析数据存储在称为方法数据对象（method data object，MDO）的结构中。

字节码解释器和 C1 编译器使用 MDO 记录 JIT 编译器在确定要进行什么优化时使用的信息。MDO 存储诸如调用的方法、选取的分支以及在调用点中观察到的类型等信息。

系统会维护一些计数器（counter），用来记录所剖析属性的“热度”（hotness），其中的值

会在剖析过程中衰减。这样可以确保方法只有在到达编译队列的头部而且仍然有足够的热度时才会被编译。

一旦收集到这种剖析数据并决定编译后，具体的细节就由各个编译器接管。编译器会为要编译的代码构建一种内部表示。内部表示的确切性质取决于使用的是哪个编译器（C1 还是 C2）。

编译器会基于这种内部表示对代码进行大量优化。HotSpot 的 JIT 编译器能够执行许多不同的现代编译器优化技术，包括：

- 内联
- 循环展开
- 逃逸分析
- 锁消除和锁合并
- 单态分发
- 内部函数
- 栈上替换

下面几节会依次进行介绍。

当遇到其中的每种技术时，一定要记住，大多数优化部分或完全依赖于运行时的信息和支持。

HotSpot 中的两个 JIT 编译器也使用了上述技术的不同子集，并且对于如何进行编译有着不同的理念。特别是，C1 不会进行**推测性优化**（speculative optimization）。这种优化的使用依赖于对执行特性的未经证实的假设。激进的优化器（比如 C2）会基于观察到的运行时执行情况做出某种假设，并根据这一假设进行优化。这种简化的假设有可能带来较大（甚至是非常大）的性能加速。

假设有可能并不正确，而且之后会失效，推测性优化总是会用“健全性检查”（sanity check）来避免这种情况发生，也就是所谓的**守卫**（guard）。守卫会确保假设仍然成立，并在每次运行优化的代码之前进行检查。

如果守卫失效，编译后的代码就不再安全，必须将其删除。HotSpot 会立即**取消优化**（deoptimize），并将该方法降级回解释模式，以防止任何不正确的代码被执行。

10.3　内联

内联（inlining）是这样的过程：选定某个被调用方法（callee），将其内容复制到被调用处，即调用点中。

这就去掉了与方法调用相关的开销，虽然不是很大，但包括如下几个方面：

- 设置要传递的参数；
- 查找要调用的精确方法；
- 为新的调用栈帧（比如局部变量和求值栈）创建新的运行时数据结构；
- 将控制转移到新方法；

- 可能要向调用者返回一个结果。

内联是 JIT 编译器所应用的最重要的优化之一，也叫**网关优化**（gateway optimization），因为它通过消除方法的边界，缩短了相关代码的距离。

考虑如下代码：

```
int result = add(a, b);

private int add(int x, int y) {
    return x + y;
}
```

内联优化会将 `add()` 的方法体复制到它的调用点中，实际上得到：

```
int result = a + b;
```

HotSpot 的内联编译器优化支持开发人员编写组织清晰、可复用的代码。它代表的理念是，开发人员不需要手动进行细微的优化。相反，HotSpot 使用自动化的统计分析来确定相关代码何时应该放在一起。内联也进而拓宽了其他优化的范围，包括：

- 逃逸分析（escape analysis）
- 死代码消除（dead code elimination，DCE）
- 循环展开（loop unrolling）
- 锁消除（lock elision）

10.3.1 内联的限制

有时虚拟机需要对内联子系统施加一些限制。比如，虚拟机可能需要控制：

- JIT 编译器优化一个方法所花的时间；
- 生成的原生代码的大小（这会进而影响代码缓存所占用的内存多少）。

如果没有这些约束，编译器可能就会陷入很深的内联调用链中不能自拔，或是会因巨大的原生方法而填满代码缓存。JIT 编译资源非常宝贵，这个一般性原则在这里再次发挥作用。

HotSpot 在决定是否内联某个方法时会考虑几个因素，包括：

- 要内联的方法的字节码大小；
- 要内联的方法在当前调用链中的深度；
- 该方法的编译版本在代码缓存中已占用的空间量。

在图 10-6 中，我们可以看到 JITWatch 以可视化的方式显示出，JIT 编译器将一个方法调用链内联到最终调用的地方，不过拒绝了内联我们有意创建的一个超过最大默认限制的方法。

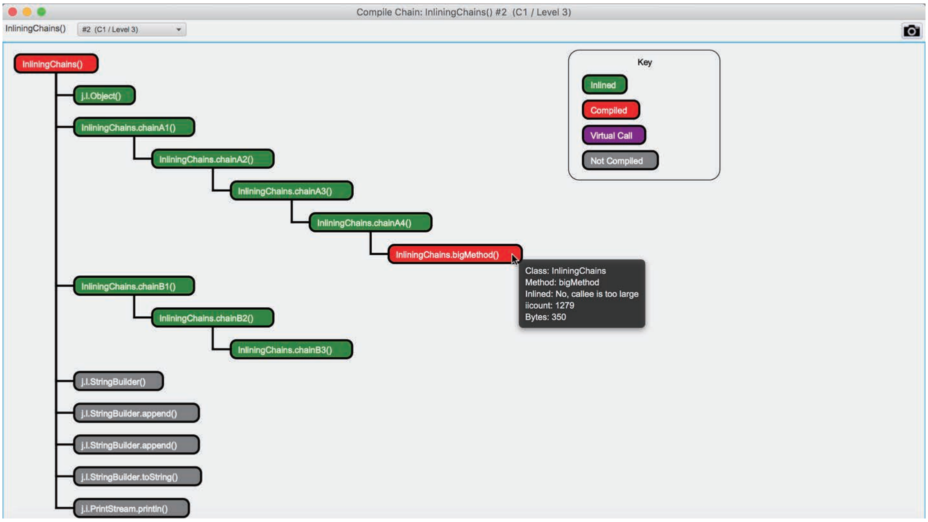


图 10-6：内联到父方法中的方法调用链

这种深度的内联非常典型，有助于拓宽优化范围。

10.3.2　调优内联子系统

在确定了所有重要的方法都已编译，并且调整了代码缓存以容纳应用程序中最显著的方法后，下一步就是考虑内联。表 10-1 显示了用于控制内联子系统行为的基本 JVM 开关。这些开关可以用来扩展 9.5 节所介绍的简单 JIT 调优技术。

表10-1：内联开关

开　　关	默认值（JDK 8，Linux x86_64）	解　　释
`-XX:MaxInlineSize=<n>`	35 字节的字节码	内联方法最大为这个值
`-XX:FreqInlineSize=<n>`	325 字节的字节码	内联“热”方法（频繁调用的方法）最大为这个值
`-XX:InlineSmallCode=<n>`	1000 字节的原生代码（非分层编译）；2000 字节的原生代码（分层编译）	如果最后一层编译在代码缓存中占用的空间已经超过这个值，不要内联方法
`-XX:MaxInlineLevel=<n>`	9	调用栈帧的深度不能超过这个值

如果有重要的方法没有被内联（比如因为“有点太大”，超过了默认限制而无法内联），那么在某些情况下，可能需要调整 JVM 参数让这些方法内联。在这些情况下，我们会从调整 `-XX:MaxInlineSize` 或 `-XX:FreqInlineSize` 入手，并验证这样的改变是否会带来可观测到的改进。

任何调整这些参数的调优工作，都必须以观测到的数据为依据。没有考虑实际数据，也没有证明修改参数的价值所在，这就是“摆弄开关”反模式的核心问题（参见 4.5.2 节）。

10.4　循环展开

一旦循环中的任何方法调用被内联（在可能的情况下），编译器就能更好地了解每次循环迭代的大小和成本。基于这些信息，它可以考虑**展开循环**（unrolling the loop），以减少执行时必须跳回到循环起点的次数。

每一次**向后跳转**都会对处理器造成不良影响，CPU 会因为这个跳转而丢弃流水线中接下来的指令。一般来说，循环体越短，向后跳转的相对成本就越高。因此，HotSpot 会根据以下标准来决定是否展开循环：

- 循环计数器变量的类型（通常是 `int` 或 `long`，而不是对象类型）；
- 循环步长（每次迭代时循环计数器如何改变）；
- 循环内出口点（`return` 或 `break`）的数量。

考虑一些示例方法，按顺序读取数组中的数据并求和。访问模式在汇编语言中就是 `[base, index, offset]` 寻址：

- `base register`（基址寄存器），包含数组中数据的起始地址；
- `index register`（索引寄存器），包含循环计数器（乘以数据类型的大小）；
- `offset`（偏移），用于在展开后的每次访问中的偏移。

```
add rbx, QWORD PTR [base register + index register * size + offset]
```

循环展开行为在不同的 HotSpot 版本之间可能会有差异，并且高度依赖于架构。

给定一个在 `long[]` 数组上执行的循环，我们可以查看循环展开的条件。当循环访问数组时，HotSpot 可以通过将循环分成 3 个部分来消除数组边界检查，如表 10-2 所示。

表10-2：消除边界检查

循环的不同部分	是否进行边界检查	解　　释
循环前	是	执行带有边界检查的初始迭代
主循环	否	使用循环步长计算在不需要边界检查的情况下能执行的最大迭代次数
循环后	是	仍然执行带有边界检查的迭代

下面是一些示例的设置代码，用于创建一个供循环迭代使用的数组：

```
private static final int MAX = 1_000_000;

private long[] data = new long[MAX];

private void createData()
{
    java.util.Random random = new java.util.Random();
```

```
    for (int i = 0; i < MAX; i++)
    {
        data[i] = random.nextLong();
    }
}
```

我们可以使用一个 JMH 基准测试来比较使用 int 计数器或 long 计数器在同样的数组上执行迭代的性能：

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class LoopUnrollingCounter
{
        private static final int MAX = 1_000_000;

        private long[] data = new long[MAX];

        @Setup
        public void createData()
        {
                java.util.Random random = new java.util.Random();

                for (int i = 0; i < MAX; i++)
                {
                        data[i] = random.nextLong();
                }
        }

        @Benchmark
        public long intStride1()
        {

                long sum = 0;
                for (int i = 0; i < MAX; i++)
                {
                        sum += data[i];
                }
                return sum;
        }

        @Benchmark
        public long longStride1()
        {
                long sum = 0;
                for (long l = 0; l < MAX; l++)
                {
                        sum += data[(int) l];
                }
                return sum;
        }
}
```

结果如下：

```
Benchmark                         Mode  Cnt     Score   Error  Units
LoopUnrollingCounter.intStride1   thrpt  200  2423.818 ± 2.547  ops/s
LoopUnrollingCounter.longStride1  thrpt  200  1469.833 ± 0.721  ops/s
```

使用 `int` 计数器的循环每秒要多执行近 64% 的操作。

如果深入到汇编代码中，就会发现使用 `long` 计数器的循环体不会被展开，并且循环中还会包含一个安全点轮询。JIT 编译器插入这些安全点检查，以减少编译后的代码在不检查安全点标志的情况下长期运行的可能性（参见 7.2.1 节）。

其他微基准测试可能会使用**可变步长**，其中增量保存在一个变量中，而不是使用一个编译时常量。使用可变步长，我们可以看到循环没有展开，而且在向后的分支之前插入了一个安全点。

循环展开小结

HotSpot 包含一些针对循环展开的具体优化：

- 它可以优化使用 `int`、`short` 或 `char` 类型计数器的计数循环；
- 它可以展开循环体，并移除安全点轮询；
- 展开循环可以减少向后分支的数量，也就减少了相关的分支预测成本；
- 移除安全点轮询可以进一步减少每次循环迭代要完成的工作。

不过，你应该始终亲自验证这些示例的效果，而不是假设它们适用于所有的架构和 HotSpot 版本。

10.5 逃逸分析

HotSpot 可以执行基于作用域的分析，以确定某个方法内完成的工作在该方法的边界之外是否可见以及是否有副作用。这种**逃逸分析**技术可以用来确定在方法内分配的对象在该方法的作用域之外是否可见。

> 在任何内联发生之后，系统会尝试进行逃逸分析优化。内联会将被调用方法的方法体复制到调用点中，这可以防止仅作为方法参数传入的对象被标记为逃逸。

在逃逸分析阶段，HotSpot 将可能会逃逸的对象分为 3 种类型。下面这段来自 hotspot/src/share/vm/opto/escape.hpp 的代码详细描述了不同的逃逸情况：

```
typedef enum {

  NoEscape      = 1, // 对象不会逃逸出方法或线程，也没有被传递给调用。
                     // 它可以用标量替换

  ArgEscape     = 2, // 对象不会逃逸出方法或线程，但是作为参数被传递给了
                     // 调用或被引用了，它在调用期间不会逃逸

  GlobalEscape  = 3  // 对象会逃逸出方法或线程
}
```

10.5.1 消除堆分配

在紧密的循环中创建新对象会给内存分配子系统带来压力。生成大量寿命很短的对象将需要频繁地执行 Minor GC 来清理它们。分配率可能非常高，以至于堆的新生代被填满，那么这些寿命短暂的对象就会过早地晋升到老年代。如果发生这种情况，就需要开销更大的整堆收集来清理它们。

HotSpot 逃逸分析优化的设计目的是让开发人员能够编写符合 Java 习惯的代码，而无须担心对象分配率问题。

通过证明一个已分配的对象不会逃逸出当前方法（被归类为 `NoEscape`），虚拟机可以应用一种叫作**标量替换**（scalar replacement）的优化。对象中的字段会变成标量值，类似于它们都是局部变量而非对象字段。然后，它们可以被一个叫作**寄存器分配器**（register allocator）的 HotSpot 组件安排到 CPU 寄存器中。

> 如果没有足够的空闲寄存器，那么标量值就可以放到当前的栈帧上（这被称为**栈溢出**，stack spill）。

逃逸分析的目的是推断出是否可以避免堆分配。如果可以，就能自动将对象分配在栈上，从而可以减少一点垃圾收集的压力。

我们来看一个对象分配的例子，因为 `MyObj` 的实例不会离开方法作用域，所以会被归类为 `NoEscape`：

```
public long noEscape() {
    long sum = 0;

    for (int i = 0; i < 1_000_000; i++) {
        MyObj foo = new MyObj(i); // foo不会逃逸出该方法，即NoEscape
        sum += foo.bar();
    }

    return sum;
}
```

下面是另一个对象分配的例子，由于 `MyObj` 的实例被作为参数传递给了 `extBar()` 方法，因此它被归类为 `ArgEscape`。

```
public long argEscape() {
    long sum = 0;

    for (int i = 0; i < 1_000_000; i++) {
        MyObj foo = new MyObj(i);
        sum += extBar(foo); // foo被作为参数传递给了extBar (ArgEscape)
    }

    return sum;
}
```

如果在执行逃逸分析之前，对 `extBar()` 的调用被内联到循环体中，那么 `MyObj` 的实例将被归类为 `NoEscape`，从而可以避免在堆上分配。

10.5.2 锁与逃逸分析

HotSpot 能够使用逃逸分析和一些相关技术来优化锁的性能。

> 这些优化仅应用于内置锁（即使用 `synchronized` 的锁），不适用于来自 `java.util.concurrent` 的锁。

可应用的主要锁优化包括：

- 移除不会逃逸的对象上的锁（**锁消除**，lock elision）；
- 合并使用同一把锁的连续锁定区域（**锁合并**，lock coarsening）；
- 检测重复获取同一把锁但是没有解锁的地方（**嵌套锁**，nested lock）。

当同一个对象上出现连续的锁时，HotSpot 会检查是否有可能扩大锁定区域。为了实现这一点，当 HotSpot 遇到锁时，它将向后搜索，看看是否有同一对象上的解锁操作。如果有，那么它就会考虑是否可以把这两个锁定区域合并起来以得到一个更大的区域。

如果想了解更多内容，可以阅读 Java 虚拟机规范。如图 10-7 所示，我们可以直接在 JITWatch 中看到这种效果。

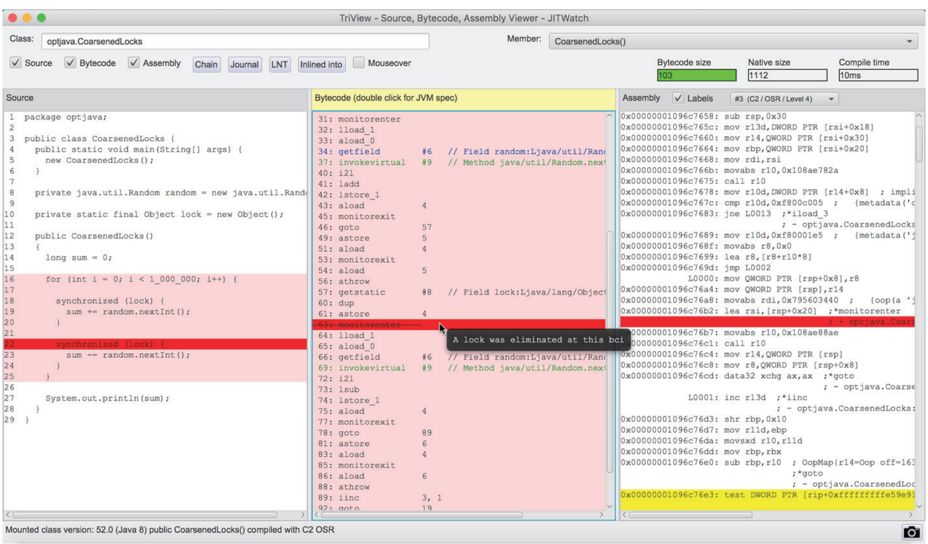


图 10-7：锁合并

锁合并优化默认是开启的，不过可以使用虚拟机开关 `-XX:-EliminateLocks` 将其禁用，以

查看其影响。

HotSpot 还能够检测到在同一对象上的嵌套锁，并移除内部的锁，因为当前线程已经在外部获取到了这把锁。



在撰写本书时，Java 8 中的嵌套锁消除似乎对声明为 `static final` 的锁和 `this` 上的锁都有效。

嵌套锁优化默认是开启的，但可以使用虚拟机开关 `-XX:-EliminateNestedLocks` 将其禁用。可以在图 10-8 中看到 JITWatch 中的嵌套锁检测。

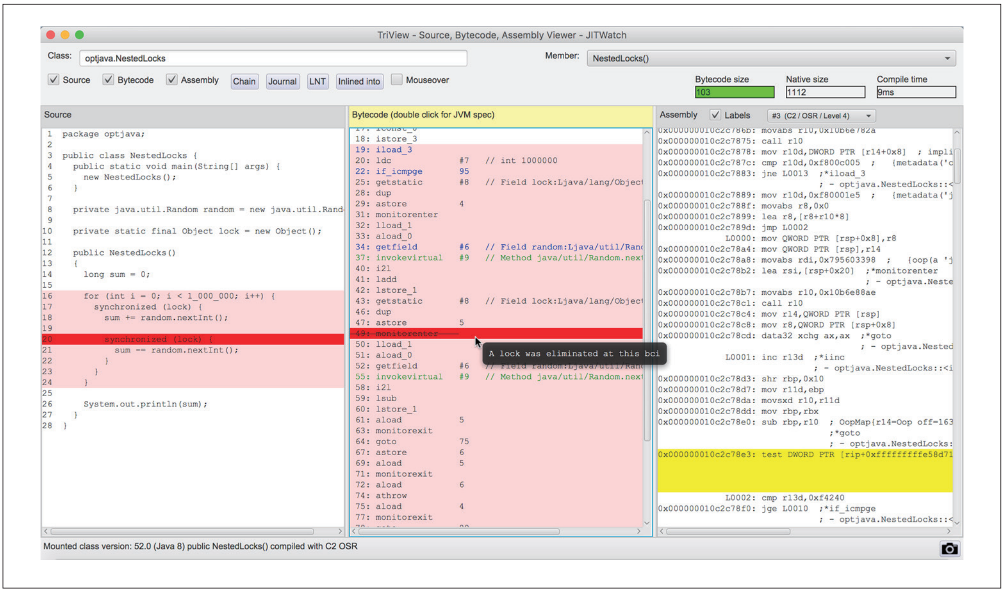


图 10-8：消除嵌套锁

HotSpot 会自动计算何时可以安全地合并或消除锁。像 JITWatch 这样的工具可以以可视化方式显示出锁被优化的位置，如果使用调试版的 JVM，可以输出与锁有关的更多信息。

10.5.3　逃逸分析的限制

和其他优化一样，逃逸分析也需要权衡，因为每一次分配，即使不发生在堆上，肯定会发生在某个地方，而 CPU 寄存器和栈空间是相对稀缺的资源。HotSpot 有一个限制，即默认情况下，超过 64 个元素的数组将无法从逃逸分析中受益。这个大小可以用如下的虚拟机开关来控制：

```
-XX:EliminateAllocationArraySizeLimit=<n>
```

考虑一个热的代码路径，其中会分配一个临时数组，用于从缓冲区读取数据。如果该数组

不会逃逸出当前方法的作用域，则逃逸分析应该阻止这次堆分配。然而，如果数组的长度超过 64 个元素（即使这些元素没有全部用到），那么它必须存储在堆上，这可能会使这段代码的分配率快速增加。

在下面的 JMH 基准测试中，测试方法会分配 3 个大小分别为 63、64、65 的不会逃逸的数组。

> 之所以要测试大小为 63 的数组，是要确保大小为 64 的数组不会只因为内存对齐的缘故而快于大小为 65 的数组。

每次测试只使用前两个数组元素 `a[0]` 和 `a[1]`，这是因为逃逸分析的限制只考虑数组长度，而不会考虑所使用的最大索引：

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class EscapeTestArraySize {

    private java.util.Random random = new java.util.Random();

    @Benchmark
    public long arraySize63() {
        int[] a = new int[63];

        a[0] = random.nextInt();
        a[1] = random.nextInt();

        return a[0] + a[1];
    }

    @Benchmark
    public long arraySize64() {
        int[] a = new int[64];

        a[0] = random.nextInt();
        a[1] = random.nextInt();

        return a[0] + a[1];
    }

    @Benchmark
    public long arraySize65() {
        int[] a = new int[65];

        a[0] = random.nextInt();
```

```
        a[1] = random.nextInt();

        return a[0] + a[1];
    }
}
```

结果显示，一旦数组分配无法从逃逸分析优化中受益，性能就会大幅下降：

```
Benchmark                          Mode  Cnt         Score       Error  Units
EscapeTestArraySize.arraySize63   thrpt  200  49824186.696 ±  9366.780  ops/s
EscapeTestArraySize.arraySize64   thrpt  200  49815447.849 ±  2328.928  ops/s
EscapeTestArraySize.arraySize65   thrpt  200  21115003.388 ± 34005.817  ops/s
```

如果你发现确实需要在热代码中分配更大的数组，那么可以让虚拟机支持优化更大的数组。将限制修改为 65 个元素，再次运行该基准测试就会显示性能恢复了：

```
$ java -XX:EliminateAllocationArraySizeLimit=65 -jar target/benchmarks.jar

Benchmark                          Mode  Cnt         Score      Error  Units
EscapeTestArraySize.arraySize63   thrpt  200  49814492.787 ± 2283.941  ops/s
EscapeTestArraySize.arraySize64   thrpt  200  49815595.566 ± 5833.359  ops/s
EscapeTestArraySize.arraySize65   thrpt  200  49818143.279 ± 2347.695  ops/s
```

另一个重要的限制是，HotSpot 不支持**部分逃逸分析**（partial escape analysis，也叫**流敏感逃逸分析**）。

> jRockit JVM 确实支持部分逃逸分析，但在这两个 JVM 合并后，该技术并没有被带到 HotSpot 中。

如果发现某个对象在任何一个分支上逃逸出了方法的作用域，那将不会应用避免在堆上分配该对象的优化。在下面的例子中，如果看到两个分支都被执行，那么对象有时候会逃逸出该方法，并且必须被归类为 ArgEscape。这将增加对象的分配率，给垃圾收集带来额外的压力。

```
for (int i = 0; i < 100_000_000; i++) {
    Object mightEscape = new Object(i);

    if (condition) {
        result += inlineableMethod(mightEscape);
    } else {
        result += tooBigToInline(mightEscape);
    }
}
```

如果在代码中有可能让对象的分配限制在一个不会逃逸的分支内，如下面的例子所示，那么你将从该路径的逃逸分析中受益。

```
for (int i = 0; i < 100_000_000; i++) {

    if (condition) {
        Object mightEscape = new Object(i);
```

```
            result += inlineableMethod(mightEscape);
        } else {
            Object mightEscape = new Object(i);
            result += tooBigToInline(mightEscape);
        }
    }
```

10.6 单态分派

HotSpot 的 C2 编译器所做的许多推测性优化是基于实验研究的。一个例子就是被称为**单态分派**（monomorphic dispatch）的技术（“monomorphic”这个词源于希腊语，意思是“单一形式”）。

它依赖于一个奇怪但很强大的观测事实：在人类编写的代码中，在任何一个单独的调用点上通常只观察到一种运行时类型是接收对象的类型。

> 这通常被理解为反映了人类设计面向对象软件的方式，而不是 Java 或 JVM 特有的东西。

也就是说，当我们在对象上调用一个方法时，如果第一次调用时检查了该对象的运行时类型，那么它很可能在以后的每次调用中都是相同的类型。

如果这个推测的假设成立，那么可以优化该点的方法调用。特别是，可以去掉在虚表（vtable）中查找方法这层间接操作。如果总是相同的类型，那么我们可以计算一次调用目标，然后用一个快速类型测试（守卫）代替 `invokevirtual` 指令，之后再跳转到编译后的方法体。

换句话说，通过 klass 指针和 vtable 进行的虚方法查找和相关间接操作只需要进行一次，然后可以缓存下来以供该调用点将来的调用使用。

> 对于一个 `invokevirtual` 调用点，我们只能看到定义了要执行的方法的基本类型以及它的任何子类型。这是里氏替换原则的另一种表现形式。

考虑如下代码：

```
java.util.Date date = getDate();
System.out.println(date.toInstant());
```

如果 `getDate()` 方法总是返回一个 `java.util.Date` 实例，那么我们就可以假设 `toInstant()` 的调用是单态的。然而如果这段代码在经过多次迭代之后，`getDate()` 突然返回一个 `java.sql.Date` 实例，那么单态假设就不再有效，因为现在必须调用 `toInstant()` 的一个完全不同的实现。

HotSpot 的解决方案是退出优化，并将调用点恢复为使用完全虚拟分派。用来保护单态调

用的守卫非常简单，就是看 klass 字是否相同，它会在每个调用指令之前进行检查，以确保不会执行错误的代码。

典型应用程序中的大量调用都是单态的。HotSpot 还有另一种优化，叫作**双态分派**（bimorphic dispatch），这种优化很少有用。通过为每个调用点缓存两个不同的 klass 字，能够以与单态分派类似的方式处理两种不同的类型。

既不是单态也不是双态的调用点被称为**复态**（megamorphic，来自希腊语，表示“多种形态”）。如果发现一个复态调用点观测到的类型只有少数几个，那你可以使用一个小技巧来重新获得一些性能提升。其工作原理是，使用 `instanceof` 检查从原始调用点“剥离”掉一些类型，只留下可以处理两个具体类型的双态调用点。

接下来看看下面这个例子：

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

interface Shape {
        int getSides();
}

class Triangle implements Shape {
        public int getSides() {
                return 3;
        }
}

class Square implements Shape {
        public int getSides() {
                return 4;
        }
}

class Octagon implements Shape {
        public int getSides() {
                return 8;
        }
}

@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class PeelMegamorphicCallsite {

        private java.util.Random random = new java.util.Random();

        private Shape triangle = new Triangle();
        private Shape square = new Square();
        private Shape octagon = new Octagon();

        @Benchmark
```

```
public int runBimorphic() {
        Shape currentShape = null;

        switch (random.nextInt(2))
        {
        case 0:
                currentShape = triangle;
                break;
        case 1:
                currentShape = square;
                break;
        }

        return currentShape.getSides();
}

@Benchmark
public int runMegamorphic() {
        Shape currentShape = null;

        switch (random.nextInt(3))
        {
        case 0:
                currentShape = triangle;
                break;
        case 1:
                currentShape = square;
                break;
        case 2:
                currentShape = octagon;
                break;
        }

        return currentShape.getSides();
}

@Benchmark
public int runPeeledMegamorphic() {
        Shape currentShape = null;

        switch (random.nextInt(3))
        {
        case 0:
                currentShape = triangle;
                break;
        case 1:
                currentShape = square;
                break;
        case 2:
                currentShape = octagon;
                break;
        }

        // 从原始调用点剥离出一个观察到的类型
        if (currentShape instanceof Triangle) {
```

```
                return ((Triangle) currentShape).getSides();
            }
            else {
                return currentShape.getSides(); // 现在只有双态了
            }
        }
    }
```

运行这个基准测试，输出如下：

```
Benchmark                                Mode  Cnt     Score    Error  Units
PeelMega...Callsite.runBimorphic        thrpt  200  75844310 ±  43557  ops/s
PeelMega...Callsite.runMegamorphic      thrpt  200  54650385 ±  91283  ops/s
PeelMega...Callsite.runPeeledMegamorphic thrpt  200  62021478 ± 150092  ops/s
```

当在调用点观察到两个实现时，就会发生双态内联。与观察到 3 个实现，并且方法分派仍然为虚拟调用的复态调用点相比，它可以每秒多执行约 38% 的操作。需要注意的是，这个对比有欠公平，因为代码的行为是不同的。

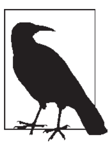
当一种观察到的类型被剥离到一个不同的调用点时，程序每秒执行的操作比复态代码多 13% 左右。

方法分派以及由此带来的性能影响是一个深刻的话题。Aleksey Shipilёv 在他的博客文章"The Black Magic of (Java) Method Dispatch"中给大家上了重要一课。

10.7　内部函数

内部函数（intrinsic）指的是一个方法的高度优化的原生实现，它们是 JVM 预先知道的，而非由 JIT 子系统动态生成。它们用于性能关键的核心方法，其功能由操作系统或 CPU 架构的特定特性来支持。这使得它们和平台相关联，并且有些内部函数可能不是每个平台都支持。

JVM 启动时，会在运行时对 CPU 进行探测，并建立一个可用处理器特性的列表。这意味着关于使用哪种优化的决策可以推迟到运行时再做，而不必在代码编译时做。



内部函数在解释器以及 C1 和 C2 JIT 编译器中都可以实现。

表 10-3 列出了一些常见内部函数的示例。

表10-3：内部函数示例

方　法	描　述
`java.lang.System.arraycopy()`	使用 CPU 上的向量支持实现更快速的复制
`java.lang.System.currentTimeMillis()`	大多数操作系统提供了快速实现
`java.lang.Math.min()`	在某些 CPU 上执行时不需要分支跳转
其他 `java.lang.Math` 方法	某些 CPU 上有直接的指令支持
加密功能（比如 AES）	硬件加速可以带来明显的性能提升

我们可以在 OpenJDK 的 HotSpot 源代码中查看内部函数模板。它们包含在 .ad 文件中（这个后缀代表的是“architecture dependent”，即架构相关）。

> Java 9 还在源代码中引入了 `@HotSpotIntrinsicCandidate` 注解来表示可能有可用的内部函数。

对于 x86_64 架构，可以在文件 hotspot/src/cpu/x86/vm/x86_64.ad 中找到它们。

例如，要计算一个数的以 10 为底的对数，可以使用 `java.lang.Math` 中的这个方法：

```
public static double log10(double a)
```

在 x86_64 架构上，该计算会用两条 FPU（浮点运算单元）指令来执行以下操作。

1. 计算常数 2 以 10 为底的对数。
2. 将上面的结果乘以参数以 2 为底的对数。

其内部函数代码如下：

```
instruct log10D_reg(regD dst) %{
  // The source and result Double operands in XMM registers
  // match(Set dst (Log10D dst));
  // fldlg2       ; push log_10(2) on the FPU stack; full 80-bit number
  // fyl2x        ; compute log_10(2) * log_2(x)
  format %{ "fldlg2\t\t\t#Log10\n\t"
            "fyl2x\t\t\t# Q=Log10*Log_2(x)\n\t"
         %}
   ins_encode(Opcode(0xD9), Opcode(0xEC),
              Push_SrcXD(dst),

              Opcode(0xD9), Opcode(0xF1),
              Push_ResultXD(dst));
  ins_pipe( pipe_slow );
%}
```

> 如果一个核心 Java 方法的源代码看上去还没有达到最优，那么可以检查一下虚拟机是否已经有一个特定于平台的内部函数实现。

当考虑添加一个新的内部函数时，你需要权衡一下额外的复杂性和内部函数有用的可能性。

例如，我们可以考虑使用内部函数来执行基本的算术运算，比如计算前 n 个数的和。传统的 Java 代码计算这个值需要的操作为 $O(n)$ 级别，但是有一个简单的公式，可以在 $O(1)$ 时间内计算它。

那么我们是否应该实现一个内部函数在常量时间内完成求和呢？

答案取决于在现实中观察到有多少个类需要计算这样的和——这样的情况并不是很多。因此，这样的内部函数用处有限，几乎可以肯定不值得在 JVM 中增加额外的复杂性。

这就强调了一点，即只有那些在真实代码中经常出现的操作，内部函数才会对性能产生很大的影响。

10.8　栈上替换

有时你会遇到在方法内包含热循环（hot loop）的代码，该方法调用的次数不足以触发编译，比如 Java 程序的 `main()` 方法。

HotSpot 仍然可以使用一种叫作**栈上替换**（on-stack replacement，OSR）的技术来优化这样的代码。这个技巧会计算解释器中向后跳转的循环分支数；当计数达到某个阈值时，这个解释的循环将被编译，执行将切换到这个编译的版本。

编译器负责确保编译版本可以使用任何状态更改，比如在解释循环内访问过的局部变量和锁。一旦编译的循环退出，那所有的状态更改必须在继续执行的位置可见。

举个例子，如果在 `main()` 方法内有这样一个热循环：

```
package optjava;

public class OnStackReplacement {
        // 调用一次的方法
        public static void main(String[] args) {
                java.util.Random r = new java.util.Random();

                long sum = 0;

                // 第一个长期运行的循环
                for (int i = 0; i < 1_000_000; i++) {
                        sum += r.nextInt(100);
                }

                // 第二个长期运行的循环
                for (int i = 0; i < 1_000_000; i++) {
                        sum += r.nextInt(100);
                }

                System.out.println(sum);
        }
}
```

该方法的字节码是这样的：

```
public static void main(java.lang.String[]);
 descriptor: ([Ljava/lang/String;)V
 flags: ACC_PUBLIC, ACC_STATIC
 Code:
   stack=4, locals=5, args_size=1
    0: new           #2      // class java/util/Random
    3: dup
    4: invokespecial #3      // Method java/util/Random."<init>":()V
    7: astore_1
    8: lconst_0
    9: lstore_2
```

```
10: iconst_0
11: istore        4
13: iload         4
15: ldc           #4      // int 1000000
17: if_icmpge     36
20: lload_2
21: aload_1
22: bipush        100
24: invokevirtual #5      // Method java/util/Random.nextInt:(I)I
27: i2l
28: ladd
29: lstore_2
30: iinc          4, 1
33: goto          13
36: getstatic     #6      // Field java/lang/System.out:Ljava/io/PrintStream;
39: lload_2
40: invokevirtual #7      // Method java/io/PrintStream.println:(J)V
43: return
```

在 33 这个位置的字节码 goto 会将控制流返回到进行循环检查的位置 13 处。

向后的分支发生的条件是，当循环到达循环体的末尾，检查其退出条件时，如果循环还没有结束，那分支会回到循环体的起点。

HotSpot 可以使用 C1 和 C2 JIT 编译器执行 OSR 编译。

JITWatch 会在字节码和源代码窗口中突出显示哪个循环是 OSR 编译的，如图 10-9 所示。

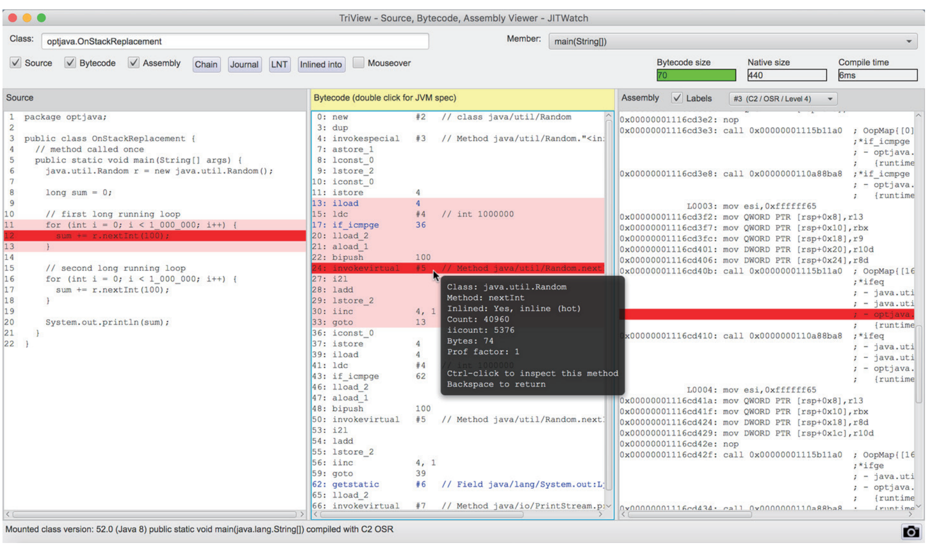


图 10-9：栈上替换

10.9　再谈安全点

在结束 JIT 编译这个主题之前，有必要列出 JVM 中要求虚拟机处于安全点的所有条件。除了 GC STW 事件之外，下面的活动也要求所有线程都处于安全点：

- 取消对一个方法的优化；
- 创建堆转储；
- 撤销偏向锁；
- 重定义一个类（比如用于注入）。

在编译后的代码中，JIT 编译器负责生成安全点检查代码（我们之前讲解循环展开时见过）；在 HotSpot 中，它会在循环里向后的分支中，以及在方法的返回处，生成安全点检查代码。

这意味着，有时线程可能需要一定的时间才会走到安全点（比如，线程正在执行一个包含大量算术运算代码而没有任何方法调用的循环）。如果这个循环被展开了，那可能要相当长一段时间才会遇到安全点。

> 只要保持程序的语义不变，JIT 编译器就可以自由地生成预测性和乱序的指令。当虚拟机到达安全点时，编译代码的状态将与程序在该点上的状态相匹配。调试器依赖于这种行为。

安全点多了，轮询检查的成本就高；安全点少了，线程就要等待较长时间才能到达安全点，这样已经到达安全点的线程就要等待其他线程到达安全点。编译器会尽量在二者之间取得平衡。

> 可以通过虚拟机开关 `-XX:+PrintGCApplicationStoppedTime` 来查看程序在安全点所花费的总时间，包括等待所有线程到达安全点的时间。结合 `-XX:+PrintSafepointStatistics` 可以了解更多关于安全点的信息。

第 13 章将再次讨论安全点，因为这是一个共性问题，对很多 JVM 子系统有影响。

10.10　核心库方法

本章最后来快速了解一下 JDK 核心库方法的大小对 JIT 编译的影响。

10.10.1　内联方法的大小上限

因为内联决策是根据方法的字节码大小做出的，所以可以通过对类文件进行静态分析来识别对于内联来说太大的方法。

开源的 JarScan 工具是 JITWatch 发布版本的一部分，启动脚本就在 JITWatch 根目录下。它可以识别一个类文件夹或一个 JAR 文件中字节码大小超过给定阈值的所有方法。

使用下面的命令，对 jre/lib/rt.jar 中的 Java 8 核心库运行该工具会发现一些有趣的结果：

```
$ ./jarScan.sh --mode=maxMethodSize \
               --limit=325 \
               --packages=java.* \
               /path/to/java/jre/lib/rt.jar
```

在 Linux x86_64 上的 Java 8u152 中，java.* 包树中有 490 个方法的字节码大小超过 325 字节（该平台的 FreqInlineSize 限制），其中一些方法会出现在热代码中，你应该不会感到奇怪。

比如 java.lang.String 类中的 toUpperCase() 和 toLowerCase() 方法，它们的字节码大小都达到了惊人的 439 字节，超出了内联的正常范围。

这两个方法之所以这么大，是因为在某些区域（locale）下，转换一个字符的大小写会改变存储它所需的 char 值的个数。因此，大小写转换方法必须能够检测到这种情况，调整字符串底层数组的大小，并复制相关内容，toUpperCase() 方法的默认实现可以证明这一点：

```java
public String toUpperCase(Locale locale) {
      if (locale == null) {
          throw new NullPointerException();
      }

      int firstLower;
      final int len = value.length;

      /* Now check if there are any characters that need to be changed. */
      scan: {
          for (firstLower = 0; firstLower < len; ) {
              int c = (int)value[firstLower];
              int srcCount;
              if ((c >= Character.MIN_HIGH_SURROGATE)
                      && (c <= Character.MAX_HIGH_SURROGATE)) {
                  c = codePointAt(firstLower);
                  srcCount = Character.charCount(c);
              } else {
                  srcCount = 1;
              }
              int upperCaseChar = Character.toUpperCaseEx(c);
              if ((upperCaseChar == Character.ERROR)
                      || (c != upperCaseChar)) {
                  break scan;
               }
               firstLower += srcCount;
          }
          return this;
      }

      /* result may grow, so i+resultOffset is the write location in result */
      int resultOffset = 0;
      char[] result = new char[len]; /* may grow */

      /* Just copy the first few upperCase characters. */
      System.arraycopy(value, 0, result, 0, firstLower);
```

```
String lang = locale.getLanguage();
boolean localeDependent =
        (lang == "tr" || lang == "az" || lang == "lt");
char[] upperCharArray;
int upperChar;
int srcChar;
int srcCount;
for (int i = firstLower; i < len; i += srcCount) {
    srcChar = (int)value[i];
    if ((char)srcChar >= Character.MIN_HIGH_SURROGATE &&
        (char)srcChar <= Character.MAX_HIGH_SURROGATE) {
        srcChar = codePointAt(i);
        srcCount = Character.charCount(srcChar);
    } else {
        srcCount = 1;
    }
    if (localeDependent) {
        upperChar = ConditionalSpecialCasing.
                               toUpperCaseEx(this, i, locale);
    } else {
        upperChar = Character.toUpperCaseEx(srcChar);
    }
    if ((upperChar == Character.ERROR)
            || (upperChar >= Character.MIN_SUPPLEMENTARY_CODE_POINT)) {
        if (upperChar == Character.ERROR) {
            if (localeDependent) {
                upperCharArray =
                        ConditionalSpecialCasing.
                                   toUpperCaseCharArray(this, i, locale);
             } else {
                 upperCharArray = Character.toUpperCaseCharArray(srcChar);
             }
         } else if (srcCount == 2) {
             resultOffset += Character.toChars(upperChar,
                                               result, i + resultOffset)
                                               - srcCount;
             continue;
         } else {
             upperCharArray = Character.toChars(upperChar);
         }

         /* Grow result if needed */
         int mapLen = upperCharArray.length;
         if (mapLen > srcCount) {
             char[] result2 = new char[result.length + mapLen - srcCount];
             System.arraycopy(result, 0, result2, 0, i + resultOffset);
             result = result2;
         }
         for (int x = 0; x < mapLen; ++x) {
             result[i + resultOffset + x] = upperCharArray[x];
         }
         resultOffset += (mapLen - srcCount);
     } else {
         result[i + resultOffset] = (char)upperChar;
     }
```

```
        }
        return new String(result, 0, len + resultOffset);
    }
```

1. 使用领域特定方法来改进性能

如果能够确定在自己的问题域中，输入的字符集不需要这种灵活性（或许输入的都是 ASCII 字符），就可以为 `toUpperCase()` 创建一个领域特定的版本，其字节码大小很容易控制在内联的限制之内。

这个特定于 ASCII 的实现，编译生成的字节码只有 69 字节：

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class DomainSpecificUpperCase {

    private static final String SOURCE =
          "The quick brown fox jumps over the lazy dog";

    public String toUpperCaseASCII(String source) {
        int len = source.length();
        char[] result = new char[len];
        for (int i = 0; i < len; i++) {
            char c = source.charAt(i);
            if (c >= 'a' && c <= 'z') {
                c -= 32;
            }
            result[i] = c;
        }
        return new String(result);
    }

    @Benchmark
    public String testStringToUpperCase() {
        return SOURCE.toUpperCase();
    }

    @Benchmark
    public String testCustomToUpperCase() {
        return toUpperCaseASCII(SOURCE);
    }
}
```

比较自定义实现与核心库的 `String.toUpperCase()` 方法的性能，结果如下：

```
Benchmark                                     Mode  Cnt     Score   Error  Units
DomainS...UpperCase.testCustomToUpperCase    thrpt  200  20138368 ± 17807  ops/s
DomainS...UpperCase.testStringToUpperCase    thrpt  200   8350400 ±  7199  ops/s
```

在这个基准测试中，领域特定版本每秒执行的操作数大约是核心库版本的 2.4 倍。

2. 小方法的好处

小方法的另一个好处是可以增加内联组合的数量。运行时数据的变化会导致代码中的不同路径变得“热”起来。

通过让方法小一点，我们可以构建不同的内联树来优化更多的热路径。对于较大的方法，可能会更早就达到内联的大小限制，从而使一些路径得不到优化。

10.10.2 编译方法的大小上限

现在演示 HotSpot 内部的另一个限制。如果方法的字节码大小超过 8000 字节，则该方法不会被编译。在生产级 JVM 中，这个限制是不能改变的，但如果运行的是调试版 JVM，就可以使用开关 `-XX:HugeMethodLimit=<n>` 来设置方法可编译的字节码的最大值。

使用如下 JarScan 命令，可以找出 JDK 核心库中大小超过编译阈值的方法：

```
./jarScan.sh --mode=maxMethodSize --limit=8000 /path/to/java/jre/lib/rt.jar
```

结果如表 10-4 所示。

表10-4：核心库中字节码大小超过编译阈值的方法

方　　法	大小（字节数）
`javax.swing.plaf.nimbus.NimbusDefaults.initializeDefaults()`	23 103
`sun.util.resources.LocaleNames.getContents()`	22 832
`sun.util.resources.TimeZoneNames.getContents()`	17 818
`com.sun.org.apache.xalan.internal.xsltc.compiler.CUP$XPathParser $actions.CUP$XPathParser$do_action()`	17 441
`javax.swing.plaf.basic,BasicLookAndFeel.initComponentDefaults()`	15 361
`com.sun.java.swing.plaf.windows.WindowsLookAndFeel.initComponentDefaults()`	14 060
`javax.swing.plaf.metal.MetalLookAndFeel.initComponentDefaults()`	12 087
`com.sun.java.swing.plaf.motif.MotifLookAndFeel.initComponentDefaults()`	11 759
`com.sun.java.swing.plaf.gtk.GTKLookAndFeel.initComponentDefaults()`	10 921
`sun.util.resources.CurrencyNames.getContents()`	8578
`javax.management.remote.rmi._RMIConnectionImpl_Tie()`	8152
`org.omg.stub.javax.management.remote.rmi._RMIConnectionImpl_Tie()`	8152

这些巨型方法都不太可能出现在热代码中。它们大多是 UI 子系统的初始化器，或者提供用于货币、国家（地区）或地域名称列表的资源。

为了说明失去 JIT 编译的影响，我们来看看对两个近乎相同的方法执行的基准测试，一个方法的字节码大小正好在 `HugeMethodLimit` 值下，另一个正好超过该阈值：

```java
private java.util.Random r = new java.util.Random();

@Benchmark
public long lessThan8000() {
    return r.nextInt() +
           r.nextInt() +
```

```
    ... // 总的字节码大小正好小于8000字节
}

@Benchmark
public long moreThan8000() {
    return r.nextInt() +
           r.nextInt() +
    ... // 总的字节码大小正好超过8000字节
}
```

结果如下：

```
Benchmark                   Mode  Cnt      Score     Error  Units
HugeMethod.lessThan8000    thrpt  100  89550.631 ±  77.703  ops/s
HugeMethod.moreThan8000    thrpt  100  44429.392 ± 102.076  ops/s
```

`moreThan8000()` 方法没有被 JIT 编译，`lessThan8000()` 方法被 JIT 编译了，前者的运行速度大约是后者的一半。有很多不创建这种巨型方法的理由（比如可读性、可维护性以及可调试性），现在又多了一个理由。

一种现实的情况是在自动生成的代码中可能会创建巨型方法。有些软件会自动生成代码来表示某个数据存储上的查询。然后查询代码会被编译，以提高在 JVM 上执行时的性能。如果查询的复杂度足够高，那么它可能达到了 HotSpot 的限制，因此使用一个像 JarScan 这样的工具来检查方法的大小可能会有所帮助。

> 虽然 JIT 编译器的很多设置可以调整，但是在调整前后都应该对自己的系统进行基准测试，因为可能存在意想不到的副作用，包括代码缓存空间、JIT 编译器队列长度，甚至垃圾收集的压力。

10.11　小结

本章介绍了 HotSpot 的 JIT 编译子系统的基础知识，并探讨了它执行的一些优化。我们了解了可以用来调优 JIT 编译器的参数，并研究了 JIT 生成的一些汇编代码。

在可操作的实用技术方面，我们可以使用 `-XX:+PrintCompilation` 标志和上一章介绍的技术来确认对各个方法的优化。"先写好代码，只在需要的时候才进行优化"，这个一般性原则在这里肯定适用。了解了内联和其他编译阈值的限制，开发人员就可以重构其代码，使其处于这些限制之内（或者在极少数情况下，修改阈值）。了解了单态分派和类型锐化的存在，意味着我们可以根据一个经典的原则来编写应用程序：设计接口，哪怕只有一个实现。这可以获得两种方法的好处——类仍然可以提供一个虚的实现，或者用于模拟测试；单一实现保留了单态分派。

第 11 章

Java语言性能技术

到目前为止，我们已经探讨了 JVM 读取开发人员写好的代码、将其转换为字节码以及进一步将其优化为高性能编译代码的机制。

如果每个 Java 应用程序都是由高质量、干净的代码组成，并且在架构时就考虑了性能，那该多好啊。然而现实并非如此。尽管这样，很多情况下 JVM 可以使用不那么理想的代码并使其工作得很好，这也证明了 JVM 这个环境的强大和健壮。

即便是代码库质量极其低下且难以维护的应用程序，通常也可以通过某种方式使其在产品中表现不错。当然，没有人想维护或修改这样的应用程序。那么当需要优化这类代码的性能时，开发人员还能做什么呢？

除了应用程序外部因素，比如网络链接、I/O 和数据库，性能的最大潜在瓶颈之一是代码的设计。设计是非常困难的一部分，没有完美的设计。

> 优化 Java 非常复杂，我们在第 13 章将介绍如何使用性能剖析工具来辅助寻找性能不理想的代码。

尽管如此，关注性能的开发人员还应该记住代码的一些基本方面，比如知道数据如何存储在应用程序中就相当重要。而且，随着业务需求的变化，数据存储方式可能也需要演进。为了解数据存储有哪些可用选项，熟悉 Java Collections API 中提供的数据结构及其实现细节对开发人员非常重要。

选择某个数据结构，却不知道它将如何被更新和查询是非常危险的。虽然有些开发人员会直接选择他们喜欢的类，使用时也不假思索，但是认真的开发人员会考虑数据将如何查询，以及哪些算法查询最高效。`java.lang.Collections` 以类方法的形式提供了很多算法和操作。

在为产品代码编写某个通用算法（比如手动实现的冒泡排序）的实现之前，先检查 `java.lang.Collections` 中是不是有可以利用的。

了解系统中的领域对象及其生命周期可以对性能产生重大影响。可以考虑一些启发式方法，应用程序内领域对象的使用方式会影响垃圾收集和增加开销，JVM 必须在运行时管理这些开销（通常是不必要的）。

本章将从关注性能的开发人员应该了解的集合类细节入手，探讨每一个问题。

11.1 优化集合

大部分编程语言库会提供至少两种通用类型的容器。

- **顺序容器**：将对象保存在某个特定位置，用索引来表示。
- **关联容器**：使用对象本身来确定它在集合中应该保存的位置。

为使特定容器的方法正常工作，被存储对象必须提供比较（comparability）和相等（equivalence）的概念。在核心的 Java Collections API 中，通常的说法是，对象必须依照 Josh Bloch 在他的 *Effective Java* 一书中推广的约定来实现 `hashCode()` 和 `equals()` 方法。

正如在 6.2 节中所看到的，引用类型的字段作为引用存储在堆中。因此，尽管我们在笼统地讨论按顺序存储的对象，但是容器中存储的并不是对象本身，而是一个指向对象的引用。这意味着你通常不会获得与使用 C/C++ 风格的数组和向量相同的性能。

这个示例说明了 Java 的托管内存子系统如何要求你放弃对内存的底层控制，以换取自动垃圾收集。放弃底层控制的常见示例是手动控制分配和回收，但是这里我们可以看到对底层内存布局的控制也被放弃了。

换一种思考方式就是，Java 至今仍然无法做到与 C 的结构体数组等价的内存布局。

Gil Tene（Azul Systems 公司的 CTO）经常把这个限制称为导致 Java 和 C 语言之间性能差距的最后一个主要障碍。ObjectLayout 网站上有更多关于内存布局如何标准化的信息，所提供的代码可以在 Java 7 及以上版本中进行编译和工作。其目的是使经过优化的 JVM 能够利用这些类型，用所描述的语义正确布局这些结构，而不破坏与其他 JVM 的兼容性。

我们在第 15 章将讨论 Java 环境的未来，并描述当前将**值类型**（value type）引入平台所做的努力。这将比对象布局代码所能做到的更进一步。

Collections API 定义了一组接口，指定了该类型的容器必须遵循的操作。图 11-1 演示的是其基本的类层次结构。

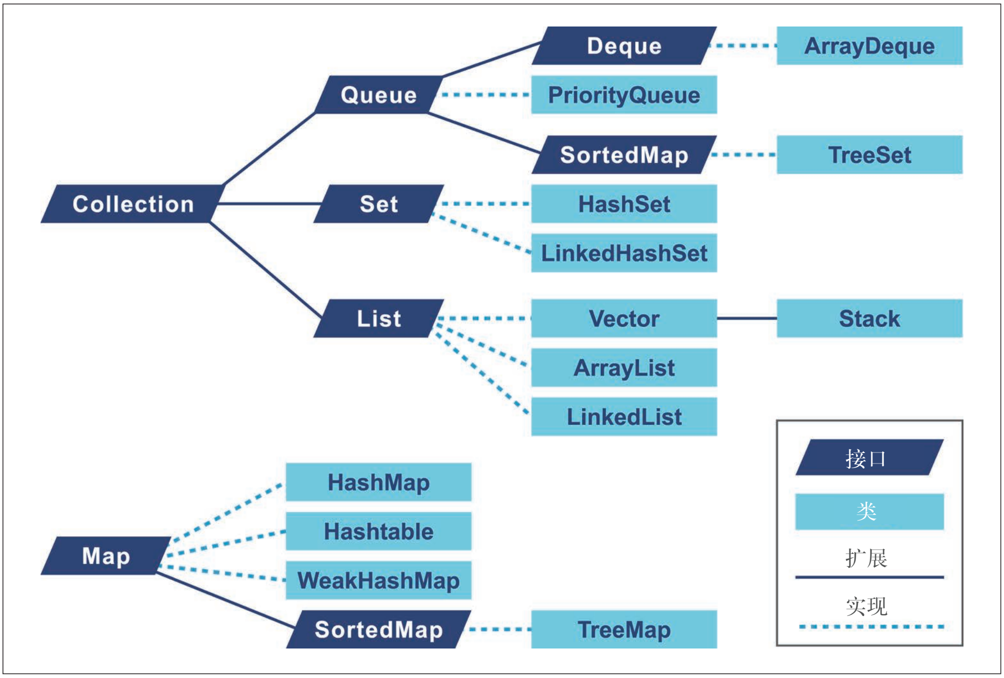


图 11-1：Java Collections API 的类层次结构

除了这些接口之外，JDK 内部还有很多种集合的实现。根据设计选择合适的实现是问题的第一部分，但我们也要认识到我们的选择可能会影响应用程序的整体性能。

11.2 针对列表的优化考虑

在核心 Java 中，基本上有两种表示列表（list）的方法，即 `ArrayList` 和 `LinkedList`。



虽然 Java 也有 `Stack` 和 `Vector` 类，但前者通常会引入并无必要的额外语义，而后者在很大程度上已经被弃用了。你的代码不应该使用 `Vector`，如果有任何地方使用了，你应该进行重构以将其删除。

下面将从基于数组实现的列表开始，依次进行讨论。

11.2.1 ArrayList

`ArrayList` 是通过一个大小固定的数组提供支持的。`ArrayList` 中可以不断加入条目，直到数量达到底层数组的最大值。当底层数组被填满时，该类将分配一个新的、更大的数组，并将原来的值复制进去。因此，有关注性能的程序员必须权衡调整大小操作的成本和不需要提前知道列表会变成多大的灵活性。一开始，`ArrayList` 底层是一个空数组。在第一次向 `ArrayList` 中增加条目时，底层会分配一个容量为 10 的数组。通过向构造器传递我们首

选的初始容量值，可以避免这次调整大小的行为。我们也可以使用 `ensureCapacity()` 来增加 `ArrayList` 的容量，以避免重新调整大小。

> 尽可能设置一个容量是明智之举，因为重新调整大小会带来性能损失。

下面是 JMH 中的一个微基准测试（在 5.2 节介绍过），演示了这一效果：

```
@Benchmark
public List<String> properlySizedArrayList() {
        List<String> list = new ArrayList<>(1_000_000);
        for(int i=0; i < 1_000_000; i++) {
                list.add(item);
        }
        return list;
}

@Benchmark
public List<String> resizingArrayList() {
        List<String> list = new ArrayList<>();
        for(int i=0; i < 1_000_000; i++) {
                list.add(item);
        }
        return list;
}
```

> 本节中的微基准测试旨在说明问题，并不是得出权威性的结论。如果你的应用程序的性能对此类操作非常敏感，那你应该考虑标准集合的替代方案。

输出如下：

```
Benchmark                             Mode  Cnt    Score     Error  Units
ResizingList.properlySizedArrayList  thrpt   10  287.388  ± 7.135   ops/s
ResizingList.resizingArrayList       thrpt   10  189.510  ± 4.530   ops/s
```

当测试插入时，`properlySizedArrayList` 测试每秒大约能够多执行 100 次操作，因为即使重新分配的成本被摊销了，仍然也会有一个总成本。尽可能选择一个大小合适的 `ArrayList`。

11.2.2 LinkedList

`LinkedList` 的增长机制更为动态化，它被实现为一个双向链表，这意味着除了其他方面之外，向列表中添加元素的时间总是 $O(1)$。每次在列表中加入新的条目就会创建一个新的节点，并让前一个条目指向它，示例如图 11-2 所示。

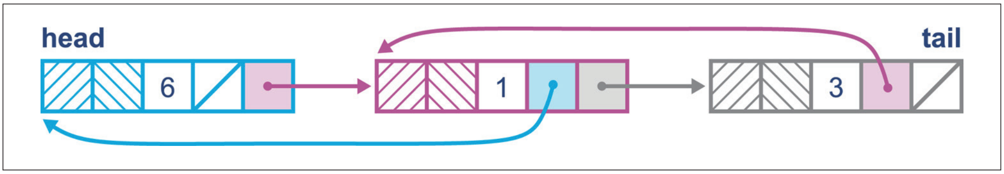


图 11-2：一个 LinkedList

11.2.3 ArrayList与LinkedList的对比

真正决定是使用 ArrayList 还是 LinkedList（或其他非标准的 List 实现）取决于访问和修改数据的模式。如果是在列表末尾插入数据，对 ArrayList 和 LinkedList 而言，就都是常量时间操作（在 ArrayList 的情况下，是在调整大小操作摊销之后）。

然而，在 ArrayList 中，要在某个索引处加入数据，则需要将从该处开始的其他所有元素向右移动一个位置。对 LinkedList 而言，则需要遍历节点引用，在列表中找到需要插入的位置，但插入本身只需要创建一个新的节点，并设置两个引用，一个指向原来位于索引处的节点的前面一个节点，一个指向原来位于索引处的节点。从下面的基准测试可以看出在两种类型的列表执行插入的性能差异：

```
Benchmark                       Mode  Cnt    Score    Error   Units
InsertBegin.beginArrayList     thrpt   10    3.402 ±  0.239  ops/ms
InsertBegin.beginLinkedList    thrpt   10  559.570 ± 68.629  ops/ms
```

列表删除也有类似的表现。从 LinkedList 中删除元素成本较低，最多需要修改两个引用。在 ArrayList 中，处于要删除位置右侧的所有元素都需要向左移动一个位置。

如果主要是随机访问列表，那么 ArrayList 是最好的选择，因为任何元素都可以在 $O(1)$ 时间内访问，而 LinkedList 则需要从开头遍历到索引位置。通过一个特定的索引来访问不同类型列表的成本可以通过以下简单的基准测试获得：

```
Benchmark                          Mode  Cnt       Score   Error   Units
AccessingList.accessArrayList     thrpt   10  269568.627 ± 0.863  ops/ms
AccessingList.accessLinkedList    thrpt   10   12972.927 ± 0.030  ops/ms
```

一般来说，除非需要 LinkedList 的特定行为，否则建议首选 ArrayList，尤其是在使用需要随机访问的算法时。尽可能提前正确设置 ArrayList 的大小，以避免重新调整所带来的损失。现代 Java 中的集合类认为开发人员应该承担所有访问的同步成本，并且在必要时使用并发集合或者手动管理同步。在 Collections 辅助类中有一个叫 synchronizedList() 的方法，它实际上是一个装饰器，将 List 方法的所有调用都封装到了一个 synchronized 块中。第 12 章将更详细地讨论如何将 java.util.concurrent 用于我们的结构，以便在编写多线程应用程序时能够更有效地使用这些结构。

11.3　针对映射的优化考虑

映射（map）通常描述键（key）和关联的值（value）之间的关系，所以就有了**关联数组**（associative array）这个术语。在 Java 中，映射遵循 `java.util.Map<K,V>` 接口。当然，键和值都必须是引用类型。

11.3.1　HashMap

Java 的 `HashMap` 在很多方面可以看作计算机科学中经典的入门级散列表（hash table），它也加入了一些额外的处理使之适合现代环境。

简化版的 `HashMap`（去掉了泛型和一些关键特性，我们后面还会再看到）有一些关键方法，如下所示：

```
public Object get(Object key) {
    // 为简化起见，不支持空的键
    if (key == null) return null;

    int hash = key.hashCode();
    int i = indexFor(hash, table.length);
    for (Entry e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }

    return null;
}

private int indexFor(int h, int length) {
    return h & (length-1);
}

// 这是一个链表节点
static class Node implements Map.Entry {
    final int hash;
    final Object key;
    Object value;
    Node next;

    Node(int h, Object k, Object v, Entry n) {
        hash = h;
        key = k;
        value = v;
        next = n;
    }
}
```

在这种情况下，`HashMap.Node` 类被限制为只能在 `java.util` 包中访问，也就是说，这是静态类的一个经典用例。

`HashMap` 的布局如图 11-3 所示。

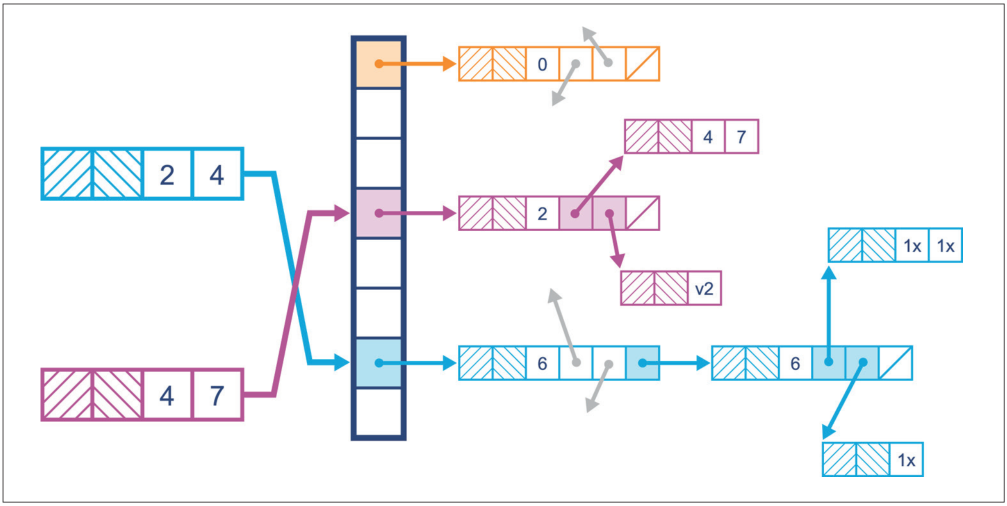


图 11-3：一个简单的 HashMap

最初桶的条目将储存在一个列表中。在查找值时，先计算出键的散列，然后用 equals() 方法在列表中找到相应的键。因为用到了这两种手段，所以 HashMap 中不允许存在重复的键。插入同样的键会替换掉当前储存在 HashMap 中的键。

现代 Java 版本中的一个改进是 indexFor() 已经被使用键对象的 hashCode() 的代码所替代，并应用了一个掩码将散列中的高位向后面移位了。

这是一种设计权衡，以确保 HashMap 在计算键散列到哪个桶时考虑到散列的高位。如果不这样做，那么在计算索引时可能就不会用到高位。这是有问题的，原因有很多，最重要的是它违反了香农的**严格雪崩准则**（strict avalanche criterion），即输入数据任意微小的变化都可能对散列函数的输出产生巨大改变。

有两个重要的变量影响着 HashMap 的性能：initialCapacity 和 loadFactor，这两个变量都可以通过传递给构造器的参数来设置。HashMap 的容量（capacity）代表当前创建的桶的数量，其默认值为 16。loadFactor 表示散列表要达到多满时，容量才会自动增加。增加容量和重新计算散列的过程叫作**再散列**（rehash），它会使可用的桶数增加一倍，并重新分布存储的数据。



设置 HashMap 的 initialCapacity 遵循与 ArrayList 相同的规则：如果事先知道大概要存储多少信息，你就应该设置它。

一个准确的 initialCapacity 会避免在表增长时自动再散列。你也可以调整 loadFactor，但 0.75 这个默认值在存储空间和访问时间之间提供了一个很好的平衡。loadFactor 大于 0.75 会降低再散列的需求，但随着桶越来越满，访问速度会变慢。将 initialCapacity 设置为元素的最大数量除以 loadFactor，将防止再散列操作的发生。

HashMap 对 get() 和 put() 操作提供了常量时间支持，但是迭代的成本可能很高。正如在 JavaDoc 中提到的，将 initialCapacity 和 loadFactor 设置得过高会严重影响迭代性能。

另一个影响性能的因素是一个称为**树化**（treeifying）的过程。这个相对较新的创新是 HashMap 的一个内部实现细节，但可能对性能工程师有些作用。

考虑桶被高度填充的情况。如果桶的元素被实现为 LinkedList，随着列表的增长，通过遍历寻找一个元素的平均成本就会越来越高。

为了抵消这种线性效应，HashMap 的现代实现采用了一个新的机制，即一旦一个桶中的元素数量达到了 TREEIFY_THRESHOLD，它就会被转换为一组 TreeNode（表现与 TreeMap 类似）。

为什么不从一开始就这样做呢？这是因为 TreeNode 的大小大约是列表节点的两倍，所以要付出空间成本。一个分布均匀的散列函数很少会导致桶被转换为 TreeNode。如果应用程序中出现了这种情况，那么是时候考虑重新访问散列函数以及 initialCapacity 和 loadFactor 的设置了。

和性能问题的所有其他方面一样，实践中采用的技术都是由权衡和实用主义驱动的，所以你应该采用类似这种脚踏实地的、数据驱动的方法来分析自己的代码。

LinkedHashMap

LinkedHashMap 是 HashMap 的一个子类，它通过所有元素之间的双向链表来维护元素的插入顺序。

LinkedHashMap 默认情况下维护插入顺序，但也可以将其模式切换为访问顺序。它经常用于用户对顺序有要求的地方，而且其成本也没有 TreeMap 那么高。

因为大多数情况下插入顺序和访问顺序对 Map 的用户不重要，所以需要正确选择 LinkedHashMap 集合的场合应该相对较少。

11.3.2 TreeMap

TreeMap 本质上是一种红黑树实现。这种类型的树基本上是一种带有额外的元数据（节点着色）的二叉树结构，其中的元数据用于防止树变得过于不平衡。

> 因为树的节点（TreeNode）需要排序，所以键必须提供一个与 equals() 方法一致的比较器。

当需要一系列键时，TreeMap 就会变得非常有用，因为它支持快速访问子映射（submap）。TreeMap 也可以用来对数据进行分区，比如从起点到某个点，或者从某个点到最后。

TreeMap 的 get()、put()、containsKey() 和 remove() 操作的性能为 log(*n*)。

在实践中，大多数情况下 HashMap 能满足我们对 Map 的要求，但考虑一个例子：使用流或 Lambda 处理 Map 中的部分内容。在这些情况下，使用一个能了解数据分区的实现（比如 TreeMap）可能更有意义。

11.3.3 缺少MultiMap

Java 没有提供 MultiMap（一个键可以关联多个值）的实现。文档中给出的理由是，这不会经常用到，而且大多数情况下可以以 Map<K, List<V>> 的形式实现。但有些开源实现为 Java 提供了 MultiMap。

11.4 针对集的优化考虑

Java 包含 3 种类型的集（set），它们与 Map 都有非常相似的性能考虑。

事实上，如果先仔细看一下 HashSet 的一个版本（为简洁起见，做了删减），就会发现它明显是用 HashMap（在 LinkedHashSet 的情况下，则是用 LinkedHashMap）来实现的：

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Serializable {
        private transient HashMap<E,Object> map;

    // 用来与底层Map中的对象关联的一个哑值
    private static final Object PRESENT = new Object();

        public HashSet() {
                map = new HashMap<>();
        }

        HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }

    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
}
```

Set 的行为不允许出现重复的值，这与 Map 中的键完全一样。在 add() 方法中，HashSet 只需要将所要插入的元素 E 当作 HashMap 中的一个键，并使用一个哑对象 PRESENT 作为与其关联的值。这样做的开销很小，因为 PRESENT 对象只需创建一次，后面可以直接引用。HashSet 的第二个受保护的构造器支持创建一个 LinkedHashMap，这个可以用来模拟同样的行为，同时记录插入顺序。HashSet 的插入、删除和包含操作的时间都为 $O(1)$，它不维护元素的顺序（除非用作 LinkedHashSet），并且迭代成本取决于 initialCapacity 和 loadFactor。

TreeSet 以类似的方式实现，利用了之前讨论的已有的 TreeMap。使用它可以保留由 Comparator 定义的键的自然顺序，使其更适合基于范围的操作和迭代操作。TreeSet 可以保证插入、删除和包含操作的时间成本为 $\log(n)$，而且它会维护元素的顺序。它对于迭代和遍历子集都非常高效，任何基于范围的操作或者考虑顺序的操作最好使用它来处理。

11.5 领域对象

领域对象（domain object）是表示对应用程序非常重要的业务概念的代码。它有很多例子，比如电商网站的 Order（订单）、OrderItem（订单项）和 DeliverySchedule（交货时间表）。

这些类型之间通常会有一定的关系（一个 Order 有多个与之关联的 OrderItem 实例），比如：

```
public class Order {
    private final long id;
    private final List<OrderItem> items = new ArrayList<>();
    private DeliverySchedule schedule;

    public Order(long id) {
        this.id = id;
    }

    public DeliverySchedule getSchedule() {
        return schedule;
    }

    public void setSchedule(DeliverySchedule schedule) {
        this.schedule = schedule;
    }

    public List<OrderItem> getItems() {
        return items;
    }

    public long getId() {
        return id;
    }
}

public class OrderItem {
    private final long id;
    private final String description;
    private final double price;

    public OrderItem(long id, String description, double price) {
        this.id = id;
        this.description = description;
        this.price = price;
    }

    @Override
    public String toString() {
        return "OrderItem{" + "id=" + id + ", description=" +
            description + ", price=" + price + '}';
    }
}

public final class DeliverySchedule {
    private final LocalDate deliveryDate;
    private final String address;
    private final double deliveryCost;

    private DeliverySchedule(LocalDate deliveryDate, String address,
            double deliveryCost) {
        this.deliveryDate = deliveryDate;
        this.address = address;
```

```
        this.deliveryCost = deliveryCost;
    }

    public static DeliverySchedule of(LocalDate deliveryDate, String address,
                    double deliveryCost) {
        return new DeliverySchedule(deliveryDate, address, deliveryCost);
    }

    @Override
    public String toString() {
        return "DeliverySchedule{" + "deliveryDate=" + deliveryDate +
            ", address=" + address + ", deliveryCost=" + deliveryCost + '}';
    }
}
```

领域类型之间存在所有权关系，如图 11-4 所示。然而，领域对象图中叶节点上的大部分数据项最终是简单的数据类型，如字符串、基本类型和 `LocalDateTime` 对象。

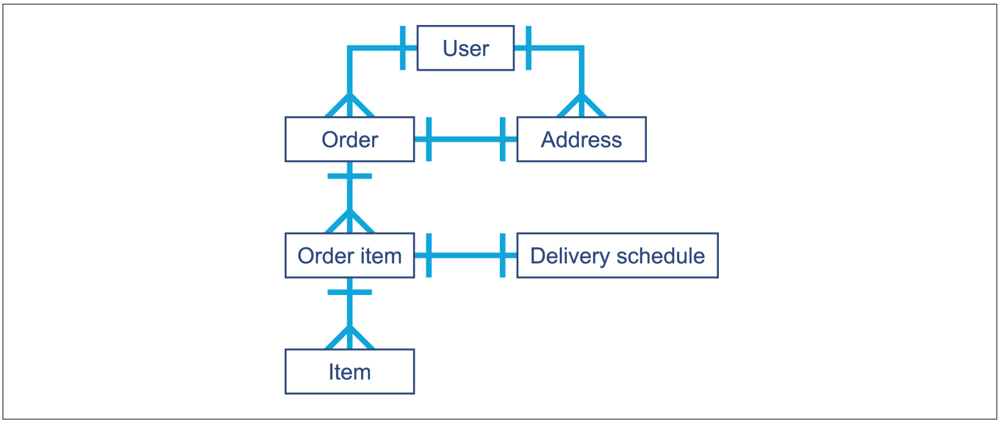


图 11-4：领域对象图

我们在 6.1 节介绍了 `jmap -histo` 命令。该命令可以帮助我们快速了解 Java 堆的状态，同样的功能也可以通过 VisualVM 这样的图形用户界面工具来获得。这些非常简单的工具都值得学习使用，因为它们有助于诊断领域对象在某些（有限但相当常见的）情况下的内存泄漏。



应用程序的领域对象地位有点特殊。因为它们表示的是应用程序中最重要的业务关注点，所以当你寻找像内存泄漏之类的错误时，它们非常显眼。

为了说明原因，需要考虑关于 Java 堆的几个基本事实。

- 分配的最常见的数据结构包括字符串、字符数组、字节数组和 Java 集合类型的实例。
- 在 `jmap` 中，与泄漏相对应的数据将表现为大得反常的数据集。

也就是说，我们希望不管是按照内存容量还是实例数目计算，排在前面的条目一般是来自核心 JDK 的数据结构。如果在 `jmap` 生成的前 30 个左右的条目中出现了特定于应用程序的

领域对象，那就可能是内存泄漏的迹象，当然这并非确定性结论。

领域对象发生泄漏的另一个常见行为是对“所有代”的影响。之所以出现这种影响，是因为特定类型的对象在应该回收的时候没被回收，从而导致它们最终会存活足够长的时间而进入 Tenured 区，在熬过足够多的收集周期后，它们的代计数可能会表现为各种可能的值。

如果用每代的字节数按照数据类型画一个直方图，那我们将看到潜在会发生泄漏的领域对象会在所有代中出现，这可能是因为它们非自然地活过了自己的自然寿命。

快速的解决办法是查看与领域对象对应的数据集的大小并检查这是否合理，是否在工作集中应该出现的领域对象的数量的预期范围之内。

另外，短寿命的领域对象可能是我们遇到的另一种形式的**浮动垃圾**问题出现的原因。回想一下针对并发收集器的 SATB 的约束——任何对象，无论生命周期多么短暂，如果它们在标记周期开始后分配，就会被认为是活的。

> 从泄漏的领域对象上偶尔还会观察到的另一个特征是，它们可能会成为导致垃圾收集标记时间很长的元凶，其根本原因是一个单个的长寿命对象会使一整个长链条上的对象存活下来。

对很多应用程序而言，领域对象就像是“煤矿里的金丝雀”，因为它们是业务关注点最明显、最自然的表示，所以似乎更容易受到内存泄漏的影响。关注性能的开发人员应该确保他们了解自己的领域和相关工作集的大小。

11.6 避免终结化

Java 的 `finalize()` 机制尝试提供自动资源管理，其方式类似于 C++ 中的**资源获取即初始化**（resource acquisition is initialization，RAII）模式。在该模式中，需要提供一个析构方法（在 Java 中就是 `finalize()`），当对象被销毁时，这个析构方法可以自动清理和释放资源。

因此，基本的使用情况非常简单。它会在创建对象时接管某个资源的所有权，并且一直持续到该对象的生命周期结束。然后当对象死亡时，该资源的所有权会被自动放弃。

> 这种模式也称为**自动资源管理**（automatic resource management，ARM）。

这种方式的标准示例是，当程序员打开一个文件句柄时，他很容易忘记在不再需要的时候调用 `close()` 函数。

接下来看一个快速而简单的 C++ 示例，它展示了如何在 C 风格的文件 I/O 之外加一层 RAII 封装：

```
class file_error {};

class file {
  public:
```

```
        file(const char* filename) : _h_file(std::fopen(filename, "w+")) {
            if (_h_file == NULL) {
                throw file_error();
            }
        }

        // 析构函数
        ~file() { std::fclose(_h_file); }

        void write(const char* str) {
            if (std::fputs(str, _h_file) == EOF) {
                throw file_error();
            }
        }

        void write(const char* buffer, std::size_t numc) {
            if (numc != 0 && std::fwrite(buffer, numc, 1, _h_file) == 0) {
                throw file_error();
            }
        }

      private:
        std::FILE* _h_file;
    };
```

这带来了良好的设计，特别当一个类型是作为某个资源（比如一个文件或网络套接字）的“持有者”存在时。在这种情况下，将资源的所有权与对象的生命周期紧密联系在一起是有意义的。这样自动释放对象的资源就成了平台的责任，而不再是程序员的责任。

11.6.1 血泪史：忘记清理

和软件领域的很多血泪史一样，这个故事始于多年来一直运行良好的生产代码。它是一个通过 TCP 连接到另一个服务来建立权限信息的服务。权限服务相当稳定，具有良好的负载均衡，通常可以立即响应请求。每当有请求时，都会打开一个新的 TCP 连接，这在设计上很不理想。

一个周末，某个变化导致权限系统的响应时间稍微慢了一些，从而使得 TCP 连接偶尔出现超时，这个代码路径之前在生产中从未出现过。系统捕获了抛出的 `TimeOutException`，没有任何日志，而且因为之前的代码中没有 `finally` 块，所以之前在正常路径下会调用的 `close()` 函数也没被调用。

不幸的是，这还不是问题的全部。没有调用 `close()` 函数意味着 TCP 连接处于打开状态。最终，应用程序所运行的生产机器耗光了所有的文件句柄，进而也影响了在该机器上运行的其他进程。解决方案是重写代码，首先关闭 TCP 连接并立即打补丁，其次建立一个连接池，不需要为每个资源打开一个新的连接。

忘记调用 `close()` 是一个很容易犯的错误，尤其是在使用专有库时。

11.6.2 为什么不使用终结化来解决这个问题

Java 最初在 `Object` 类上提供了 `finalize()` 方法，它默认不执行任何操作（通常应该保持

这种方式）。但是，它可以覆盖 `finalize()` 并提供某种行为。JavaDoc 对此描述如下。

> 当垃圾收集器确定没有更多引用指向该对象时，调用该方法。子类通过覆盖 `finalize()` 方法来释放系统资源或执行其他清理。

之所以这样实现，是因为 JVM 的垃圾收集器作为一个子系统可以明确说明对象已经死亡。如果某个类型提供了 `finalize()` 方法，那么该类型的所有对象都会被特殊对待。覆盖了 `finalize()` 方法的对象会被垃圾收集器特殊处理。JVM 的实现方式是，当调用 `java.lang.Object` 的构造器代码（任何对象的调用路径上一定会在某个点调用它）正常返回时，为其注册单独的可终结对象。

这里 HotSpot 需要注意的一个细节是，除了标准的 Java 指令外，虚拟机还有一些特殊的、与实现相关的字节码。这些特殊的字节码用于重写标准的字节码以应对某些特殊情况。

由字节码定义的完整列表，包括标准的 Java 字节码和 HotSpot 专用的字节码，都可以在 hotspot/share/interpreter/bytecodes.cpp 中找到。就目的而言，只需要关注专用的 `return_register_finalizer` 字节码，这是因为 JVMTI 可能重写 `Object.<init>()` 的字节码。为了严格遵循标准，必须确定 `Object.<init>()` 完成的点（而不用重写代码），并且使用专用的字节码来标记这个点。

在 HotSpot 解释器中可以看到实际将对象注册为需要终结化的代码。文件 src/hotspot/cpu/x86/c1_Runtime.cpp 中包含了 HotSpot 解释器 x86 版本的核心部分。这必定是与处理器相关的，因为 HotSpot 大量使用了低级的汇编 / 机器代码。`register_finalizer_id` 分支中包含了注册代码。

一旦对象被注册为需要终结化，那它在垃圾收集周期中就不会被立即回收，而会遵循以下这个扩展的生命周期。

1. 可终结对象被移到一个队列中。
2. 在应用线程重新启动之后，独立的终结线程会处理该队列，运行每个对象上的 `finalize()` 方法。
3. 一旦 `finalize()` 方法结束，对象就可以在下一个周期进行真正回收。

总之，所有要被终结化的对象必须首先让垃圾收集标记识别为不可达，然后被终结化，最后必须再次运行垃圾收集，才能完成数据收集。这意味着终结化对象至少要多进行一次额外的垃圾收集周期。对于已经进入 Tenured 区的对象而言，时间可能会相当长。终结队列的处理过程如图 11-5 所示。

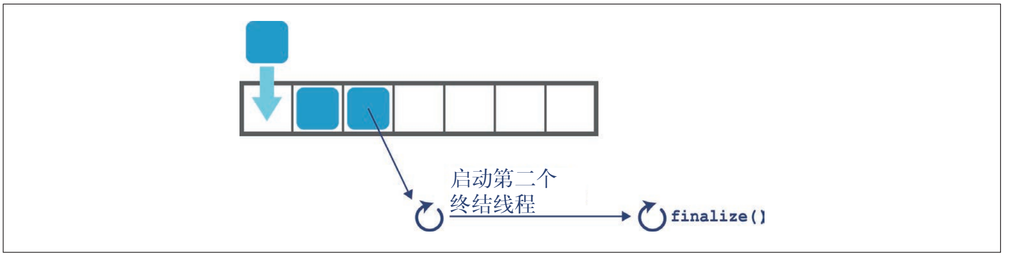


图 11-5：终结队列的处理过程

finalize() 方法还存在其他问题。比如，如果它在被终结线程执行时抛出异常会发生什么情况？这时因为用户的应用程序代码中没有上下文，所以该异常会被直接忽略。开发人员无从知道，也没有办法从终结化引起的故障中恢复。

此外，终结化过程也可能包含一个阻塞操作，因此需要 JVM 生成一个线程来运行 finalize() 方法，而创建和运行新的线程则会带来固有开销。此外，虽然线程的创建和管理不在开发人员的控制范围之内，但为了避免锁住整个 JVM 子系统，它又是必要的。

终结化的大部分实现实际上是在 Java 中进行的。JVM 有独立的线程来执行终结化，该线程与应用程序线程同时运行，完成所需要的大部分工作。核心功能包含在包私有类 java.lang.ref.Finalizer 中，读起来应该相当简单。

从这个类中可以看到运行时是如何向特定的类授予额外权限的。比如，其中包含了像这样的代码：

```
/* Invoked by VM */
static void register(Object finalizee) {
    new Finalizer(finalizee);
}
```

当然，在普通的应用程序代码中，这么写是没有意义的，因为它创建了一个未使用的对象。除非构造器有副作用（而在 Java 中这通常会被认为是一个糟糕的设计决策），否则这将毫无用处。在这种情况下，其意图是“钩住”一个新的可终结对象。

终结化的实现也非常依赖 FinalReference 类。它是 java.lang.ref.Reference 的一个子类，并且是在运行时作为特殊情况能够感知的类。与更出名的软引用和弱引用一样，FinalReference 对象也会被垃圾收集子系统特殊对待，其中包含一种支持虚拟机和 Java 代码（包括平台和用户）之间进行有趣交互的机制。

然而，由于 Java 和 C++ 这两种语言的内存管理机制之间存在差别，因此尽管有技术价值，Java 的实现仍存在致命缺陷。在 C++ 的情况下，动态内存是手动处理的，在程序员的控制之下显式地管理对象的生命周期。这意味着当对象被删除时，析构就会发生，因此资源的获取和释放直接与对象的生命周期有关。

Java 的内存管理子系统是一个根据需要运行的垃圾收集器，在用于分配的可用内存耗尽时做出响应。因此，它运行的时间间隔是不确定的，而 finalize() 方法只有当对象被收集时才会运行，这个时间也是未知的。

换句话说，终结化并不能安全地实现自动资源管理，因为垃圾收集器的运行没有任何时间保证。这意味着在这个机制中，没有任何东西能将资源的释放与对象的生命周期联系起来，所以资源总可能被耗尽。

终结化不符合其最初的主要目的。多年来，Oracle（和 Sun）给开发人员的建议是，避免在普通的应用程序代码中使用终结化，而且 Object.finalize() 在 Java 9 中已经被废弃了。

11.6.3　try-with-resources

在 Java 7 之前，关闭资源的责任完全掌握在开发人员手中。正如在 11.6.1 节中所讨论的，

我们很容易忘记清理，直到生产问题出现时才注意到其影响。下面的代码示例演示了在 Java 7 之前开发人员的责任：

```
public void readFirstLineOld(File file) throws IOException {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(file));
        String firstLine = reader.readLine();
        System.out.println(firstLine);
    } finally {
        if (reader != null) {
            reader.close();
        }
    }
}
```

开发人员必须做到以下几点。

1. 创建 `BufferedReader`，并将其初始化为 `null`，以确保它在 `finally` 块中可见。
2. 抛出，或捕捉并处理 `IOException`（可能还有它隐藏的 `FileNotFoundException`）。
3. 执行与外部资源交互的业务逻辑。
4. 检查 `reader`，确定它不是 `null` 时，关闭资源。

这个示例只使用了一个外部资源，但在处理多个外部资源时，复杂性就会大大增加。如果需要提醒，可以查看原始的 JDBC 调用。

`try`-with-resources 是 Java 7 中添加的语言级构造，支持在 `try` 关键字之后的括号中说明要创建的资源。任何实现了 `AutoCloseable` 接口的对象都可以用于 `try` 后的括号中。在 `try` 块的作用域结束时，`close()` 方法将被自动调用，不再需要开发人员必须记住去调用该函数。下面对 `close()` 方法的调用表现与前面的代码示例一样，无论业务逻辑中是否会抛出异常，都会运行：

```
public void readFirstLineNew(File file) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        String firstLine = reader.readLine();
        System.out.println(firstLine);
    }
}
```

`javap` 可以用来比较两个版本生成的字节码，下面是第一个示例中的字节码：

```
public void readFirstLineOld(java.io.File) throws java.io.IOException;
  Code:
     0: aconst_null
     1: astore_2
     2: new           #2  // class java/io/BufferedReader
     5: dup
     6: new           #3  // class java/io/FileReader
     9: dup
    10: aload_1
    11: invokespecial #4  // Method java/io/FileReader."<init>":
                          // (Ljava/io/File;)V
```

```
    14: invokespecial #5  // Method java/io/BufferedReader."<init>":
                          // (Ljava/io/Reader;)V
    17: astore_2
    18: aload_2
    19: invokevirtual #6  // Method java/io/BufferedReader.readLine:
                          // ()Ljava/lang/String;
    22: astore_3
    23: getstatic     #7  // Field java/lang/System.out:Ljava/io/PrintStream;
    26: aload_3
    27: invokevirtual #8  // Method java/io/PrintStream.println:
                          // (Ljava/lang/String;)V
    30: aload_2
    31: ifnull        54
    34: aload_2
    35: invokevirtual #9  // Method java/io/BufferedReader.close:()V
    38: goto          54
    41: astore        4
    43: aload_2
    44: ifnull        51
    47: aload_2
    48: invokevirtual #9  // Method java/io/BufferedReader.close:()V
    51: aload         4
    53: athrow
    54: return
  Exception table:
     from    to  target type
        2    30    41   any
       41    43    41   any
```

与 try-with-resources 版本等价的字节码如下：

```
public void readFirstLineNew(java.io.File) throws java.io.IOException;
  Code:
     0: new           #2  // class java/io/BufferedReader
     3: dup
     4: new           #3  // class java/io/FileReader
     7: dup
     8: aload_1
     9: invokespecial #4  // Method java/io/FileReader."<init>":
                          // (Ljava/io/File;)V
    12: invokespecial #5  // Method java/io/BufferedReader."<init>":
                          // (Ljava/io/Reader;)V
    15: astore_2
    16: aconst_null
    17: astore_3
    18: aload_2
    19: invokevirtual #6  // Method java/io/BufferedReader.readLine:
                          // ()Ljava/lang/String;
    22: astore        4
    24: getstatic     #7  // Field java/lang/System.out:Ljava/io/PrintStream;
    27: aload         4
    29: invokevirtual #8  // Method java/io/PrintStream.println:
                          // (Ljava/lang/String;)V
    32: aload_2
    33: ifnull        108
    36: aload_3
```

```
 37: ifnull        58
 40: aload_2
 41: invokevirtual #9  // Method java/io/BufferedReader.close:()V
 44: goto          108
 47: astore        4
 49: aload_3
 50: aload         4
 52: invokevirtual #11 // Method java/lang/Throwable.addSuppressed:
                       // (Ljava/lang/Throwable;)V
 55: goto          108
 58: aload_2
 59: invokevirtual #9  // Method java/io/BufferedReader.close:()V
 62: goto          108
 65: astore        4
 67: aload         4
 69: astore_3
 70: aload         4
 72: athrow
 73: astore        5
 75: aload_2
 76: ifnull        105
 79: aload_3
 80: ifnull        101
 83: aload_2
 84: invokevirtual #9  // Method java/io/BufferedReader.close:()V
 87: goto          105
 90: astore        6
 92: aload_3
 93: aload         6
 95: invokevirtual #11 // Method java/lang/Throwable.addSuppressed:
                       // (Ljava/lang/Throwable;)V
 98: goto          105
101: aload_2
102: invokevirtual #9  // Method java/io/BufferedReader.close:()V
105: aload         5
107: athrow
108: return
Exception table:
   from    to  target type
      40    44    47   Class java/lang/Throwable
      18    32    65   Class java/lang/Throwable
      18    32    73   any
      83    87    90   Class java/lang/Throwable
      65    75    73   any
```

从表面上看，try-with-resources 只是一种可以自动生成样本代码的编译器机制。然而，当坚持使用时，你会发现它是一个非常有用的简化机制，可以让类不必知道如何释放和清理其他类，进而能够更好地封装和编写没有错误的代码。

要实现与 C++ RAII 模式类似的功能，推荐使用 try-with-resources 机制。它确实把使用模式限制为基于块作用域的代码，但这是由于 Java 平台缺乏对对象生命周期的底层可见性。Java 开发人员在处理资源对象时必须遵循原则，并尽可能限制其作用域——这本身就是一个很好的设计实践。

现在我们应该清楚，尽管这两种机制（终结化和 `try`-with-resources）有着相同的设计意图，但彼此之间有着本质的区别。

终结化依赖于远在运行时内部的汇编代码来注册用于特殊垃圾收集行为的对象。然后，它使用垃圾收集器来启动清理，该清理借助于一个引用队列和单独的专用终结线程。特别是，终结化在字节码中几乎没有任何痕迹，而且这个功能是由虚拟机内部的特殊机制提供的。

相比之下，`try`-with-resources 则是一个纯粹的编译时特性，可以将其视为语法糖，只是简单地产生常规的字节码，没有其他特殊的运行时行为。使用 `try`-with-resources 唯一可能产生的性能影响是，由于它会导致自动生成大量的字节码，因此可能会影响 JIT 编译器有效地内联或编译使用这种机制的方法的能力。

但是，和所有其他潜在的性能影响一样，工程师应该衡量使用 `try`-with-resources 对运行时性能产生的影响，只有能够明确看出该特性正在引发问题，才需要花费精力进行重构。

总之，对于资源管理和几乎其他所有情况，终结化都不合适。终结化依赖于垃圾收集，而垃圾收集本身就是一个不确定的过程。这意味着任何依赖终结化的东西都没有关于资源何时被释放的时间保证。

无论最终是否会移除已经被废弃的终结化，我们的建议都是一样的：不要编写覆盖 `finalize()` 的类，如果自己的代码中有任何这样的类，那就进行重构。

11.7 方法句柄

第 9 章介绍了 `invokedynamic`。这是 Java 平台在 Java 7 中引入的一个主要进展，可以更灵活地确定在调用点要执行哪个方法。需要注意的是，`invokedynamic` 调用点直到运行时才能确定调用哪个方法。

相反，当调用点通过解释器到达时，会调用一个特殊的辅助方法，即所谓的**引导方法**（bootstrap method，BSM）。BSM 会返回一个对象，该对象表示在这个调用点应该调用的实际方法。它被称为**调用目标**（call target），被**加入**到了这个调用点中。

> 在最简单的情况下，调用目标只需要查找一次，那就是第一次遇到这个调用点时；但遇到更复杂的情况时，调用点可能会失效，所以需要重新查找（有可能生成不同的调用目标）。

一个关键的概念是**方法句柄**（method handle），它是一个对象，表示应该从 `invokedynamic` 调用点调用的方法。这与反射中的概念有点类似，但反射有些固有的限制，使其不适合用于 `invokedynamic`。

相反，Java 7 增加了一些新的类和包（特别是 `java.lang.invoke.MethodHandle`）来表示指向方法的**直接可执行**的引用。这些方法句柄对象有一组存在关联的方法，也就是可用于执行其底层的方法。其中 `invoke()` 是最常见的，但也有一些附加的辅助方法，以及与主要的调用（invoker）方法稍有区别的变种。

就像反射调用一样，方法句柄的底层方法可以有任何签名。因此，出现在方法句柄上的调

用方法的签名应该非常宽松，以便提供充分的灵活性。不过方法句柄还有一个反射所不具备的新特性。

为了解这个新特性是什么以及它为什么重要，现在来看一段以反射方式调用方法的简单代码：

```
Method m = ...
Object receiver = ...
Object o = m.invoke(receiver, new Object(), new Object());
```

生成的字节码也不足为奇：

```
17: iconst_0
18: new           #2  // class java/lang/Object
21: dup
22: invokespecial #1  // Method java/lang/Object."<init>":()V
25: aastore
26: dup
27: iconst_1
28: new           #2  // class java/lang/Object
31: dup
32: invokespecial #1  // Method java/lang/Object."<init>":()V
35: aastore
36: invokevirtual #3  // Method java/lang/reflect/Method.invoke
                      // :(Ljava/lang/Object;[Ljava/lang/Object;)
                      // Ljava/lang/Object;
```

iconst 和 aastore 操作码用于将可变参数的第 0 个和第 1 个元素存储到一个数组中，然后传给 invoke()。很明显，之后字节码调用的整体签名就是 invoke:(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;，因为该方法接受一个对象参数（方法的接收者），后面跟的是要传递给反射调用的数量可变的参数。它最终会返回一个 Object，所有这些都表明在编译时我们对这个方法调用一无所知，而是要等到运行时才知道它的各个方面。

因此，这是一个非常通用的调用，如果接收者和 Method 对象不匹配，或者参数列表不正确，那它就可能在运行时失败。

作为对比，接下来看一个用方法句柄来实现的简单示例：

```
MethodType mt = MethodType.methodType(int.class);
MethodHandles.Lookup l = MethodHandles.lookup();
MethodHandle mh = l.findVirtual(String.class, "hashCode", mt);

String receiver = "b";
int ret = (int) mh.invoke(receiver);
System.out.println(ret);
```

调用有两个部分：首先是方法句柄的查找，然后是其调用。在真实的系统中，这两个部分在时间或代码位置上距离可能非常远。方法句柄是不可变的稳定对象，可以轻松地缓存下来并持有，供以后使用。

虽然查找机制看起来像是额外的样板代码，但它是用来纠正反射从诞生之初就一直存在的一个问题——访问控制。

当一个类初始加载时，会广泛地检查其字节码，包括确认这个类不会恶意尝试去调用任何它无法访问的方法。任何这样的尝试都会导致类加载过程失败。

出于性能方面的考虑，一旦类被加载后，就不会再进行进一步的检查了。因此，我们可以尝试利用反射代码。由于多方面的原因，反射子系统（早在 Java 1.1 中就有了）最初所做的设计选择并没有让人完全满意。

方法句柄 API 采取了一种不同的方法：查找上下文。要使用它，需要通过调用 `MethodHandles.lookup()` 来创建一个上下文对象。返回的不可变对象中的状态记录了在这个上下文对象被**创建**的地方，哪些方法和字段是可以访问的。

这意味着，上下文对象既可以立即使用，也可以保存下来。这种灵活性支持选择性地访问某个类的私有方法（通过缓存一个 `Lookup` 对象，并过滤掉它的访问）。相比之下，反射只能生硬地设置 `setAccessible()`，这完全破坏了 Java 访问控制的安全特性。

现在来看看方法句柄示例中查找部分的字节码：

```
0: getstatic     #2  // Field java/lang/Integer.TYPE:Ljava/lang/Class;
3: invokestatic  #3  // Method java/lang/invoke/MethodType.methodType:
                     // (Ljava/lang/Class;)Ljava/lang/invoke/MethodType;
6: astore_1
7: invokestatic  #4  // Method java/lang/invoke/MethodHandles.lookup:
                     // ()Ljava/lang/invoke/MethodHandles$Lookup;
10: astore_2
11: aload_2
12: ldc           #5  // class java/lang/String
14: ldc           #6  // String hashCode
16: aload_1
17: invokevirtual #7  // Method java/lang/invoke/MethodHandles$Lookup.findVirtual:
                      // (Ljava/lang/Class;Ljava/lang/String;Ljava/lang/invoke/
                      // MethodType;)Ljava/lang/invoke/MethodHandle;
20: astore_3
```

这段代码生成了一个上下文对象，能够查看在 `lookup()` 静态调用发生的位置可以访问的每个方法。这里我们可以使用 `findVirtual()` 和相关方法来获取在该点上可见的任何方法的句柄。如果试图通过查找上下文来访问一个不可见的方法，那么就会抛出 `IllegalAccessException`。与反射不同，程序员无法破坏或关闭这个访问检查。

在示例中，我们只是在 `String` 上查找公开的 `hashCode()` 方法，不需要特殊访问。然而，我们仍然必须使用查找机制，平台也依旧会检查这个上下文对象是否可以访问所请求的方法。接下来看看调用方法句柄产生的字节码：

```
21: ldc           #8  // String b
23: astore        4
25: aload_3
26: aload         4
28: invokevirtual #9  // Method java/lang/invoke/MethodHandle.invoke
                      // :(Ljava/lang/String;)I
31: istore        5
33: getstatic     #10 // Field java/lang/System.out:Ljava/io/PrintStream;
36: iload         5
38: invokevirtual #11 // Method java/io/PrintStream.println:(I)V
```

这与反射的情况有本质的不同，因为对 `invoke()` 的调用不是可以接受任何参数的通用调用，相反，它描述了在运行时应该调用的方法的预期签名。

> 与相应的反射调用情况相比，方法句柄调用的字节码包含了更好的、关于调用点的静态类型信息。

在这个示例中，调用签名是 `invoke:(Ljava/lang/String;)I`，而在 `MethodHandle` 的 JavaDoc 中，没有任何信息表明该类有这样的一个方法。

相反，`javac` 源代码编译器为这个调用推导出了一个合适的类型签名，并生成了相应的调用字节码，即使 `MethodHandle` 上没有这样的方法。`javac` 生成的字节码也设置了栈，这样该调用将以通常的方式被分派（假设它可以被链接），而不需要将任何可变参数装箱到数组中。

任何加载这个字节码的 JVM 运行时都需要按照这样的方式来链接该方法调用：期望该方法句柄能在运行时表示对正确签名方法的调用，并且 `invoke()` 调用实质上将被替换为对底层方法的委托调用。

> Java 语言的这个略显奇怪的特性被称为**签名多态性**（signature polymorphism），只适用于方法句柄。

当然，这是一个非常不像 Java 语言的特性，而且使用场景有意偏向语言和框架实现者（比如 C# 的 `dynamic` 特性）。

对于许多开发人员来说，如果要简单地理解方法句柄，那就是它们提供了与核心的反射机制类似的功能，但是是以一种现代方式来实现的，能最大程度地保证静态类型安全。

11.8　小结

本章讨论了标准 Java 集合 API 的一些性能问题，以及处理领域对象的关键关注点。

最后我们探讨了另外两个与平台级别关系密切的应用程序性能方面的考虑：终结化和方法句柄。虽然很多开发人员在日常工作中并不会遇到这两个概念，但对于关注性能的工程师来说，了解和认识它们可以充实自己的技术工具箱。

下一章将继续讨论几个重要的开源库，包括那些为标准集合类提供替代选择的库，以及日志和相关问题。

第12章

并发性能技术

在迄今为止的计算历史上，软件开发人员通常以顺序格式编写代码。程序设计语言和硬件一般只提供一次处理指令的能力。许多情况下，人们享受到了所谓的“免费午餐”就是购买最新的硬件来提高应用程序的性能。芯片上可用的晶体管数量的增加带来了处理指令性能更好、更强的处理器。

许多读者都曾遇见过这样的情况：将软件搬到一个更大或更新的机器上就能解决容量问题，而不用花钱去查找底层问题或考虑不同的编程范式。

摩尔定律最初预言，芯片上的晶体管数量每年大约会增加一倍。后来，这一预测被细化到每 18 个月增加一倍。摩尔定律坚持了大约 50 年，其发展势头越来越难以维持，已经开始动摇了。图 12-1 显示了技术耗尽的影响，它也是 Herb Sutter 撰写的“The Free Lunch Is Over”一文中的核心思想，该文章恰如其分地描述了现代性能分析时代的到来。[1]

注 1：Herb Sutter，“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software，”Dr. Dobb’s Journal 30（2005）, 202–210。

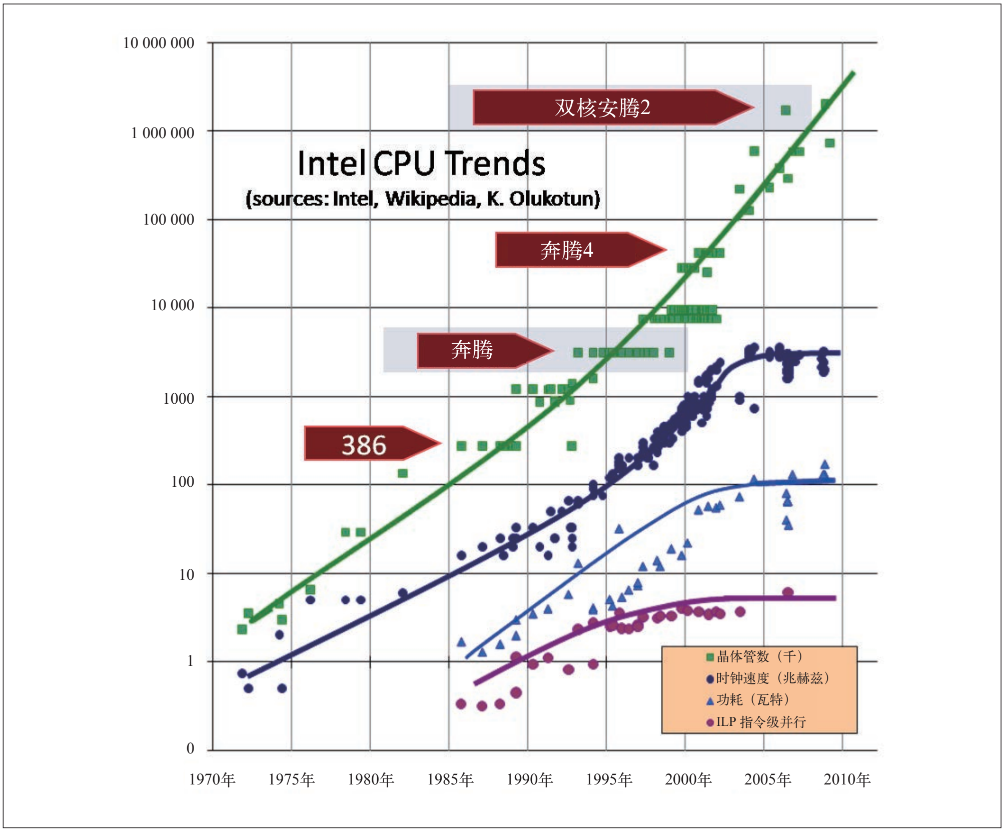


图 12-1：“The Free Lunch Is Over”（Sutter, 2005）

我们现在生活在一个多核处理器普遍存在的世界里。编写得好的应用程序可以（而且越来越多地必须）利用在多核上分配应用程序处理的优势。像 JVM 这样的应用程序执行平台具有明显的优势，因为总有一些虚拟机线程可以利用多个处理器核心进行 JIT 编译等操作。因此，即使只有一个应用程序线程的 JVM 应用程序也能从多核中受益。

为了充分利用当前的硬件，现代的 Java 专业开发人员必须至少具备关于并发及其对应用程序性能的影响的基础知识。本章只是对其进行一个基本的概述，并不打算介绍 Java 并发的全部内容。相反，除了本章的讨论之外，还应该参考 Brian Goetz 等人编写的《Java 并发编程实践》等指南。

12.1　并行介绍

单核速度在近 50 年的时间里不断得到提高，大约在 2005 年，它开始在约 3 GHz 的时钟速度上趋于平稳。然而在今天的多核世界里，**Amdahl 定律**已经成为提高计算任务执行速度的一个主要考虑因素。

我们在 1.5 节介绍了 Amdahl 定律，本章会对其进行更详细地描述。考虑将一个计算任务

分为两个部分，一部分可以并行执行，另一部分必须串行运行（例如，整理结果或调度工作单位进行并行执行）。

我们把串行部分称为 `S`，把任务所需的总时间称为 `T`。因为可以使用尽可能多的处理器来处理任务，所以把处理器的数目表示为 `N`，这意味着我们应该把 `T` 作为处理器数量的函数 `T(N)`。同时，工作的并发部分称为 `T-S`，如果这部分可以在 `N` 个处理器之间平均分配，那么这个任务所需的总时间为：

```
T(N) = S + (1/N) * (T - S)
```

这意味着无论使用多少个处理器，总耗费的时间也绝不能小于串行时间。所以如果串行开销占总开销的 5%，那么无论使用多少个核心，有效的加速时间也永远不会超过 20 倍。这个见解和公式是第 1 章介绍 Amdahl 定律背后的基本理论，在图 12-2 中可以通过另一种方式看出它的影响。

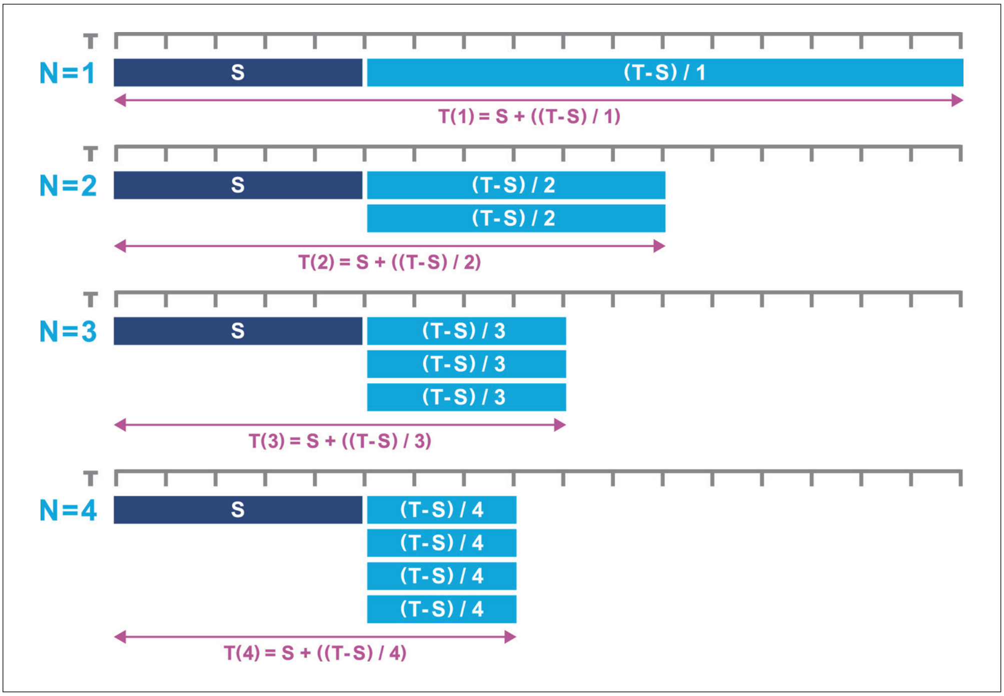


图 12-2：重新审视 Amdahl 定律

只有单线程性能的提高，比如更快的核心，才能降低 `S` 的价值。不幸的是，现代硬件的发展趋势意味着 CPU 的时钟速度不会再有任何有意义的提高。由于单核处理器不再能变得更快，因此 Amdahl 定律实际上已经是软件扩展的极限。

Amdahl 定律的一个推论是，如果并行任务之间不需要通信或其他顺序处理，那么理论上是可以无限提升速度的。这类工作负载被称为**自然并行**（embarrassingly parallel），在这种情况下，并发处理相当直接。

通常的方法是在不共享任何数据的情况下，在多个工作线程之间细分工作负载。一旦线程之间引入了共享状态或数据，工作负载的复杂性就会增加，并且不可避免地会重新引入一些串行处理和通信开销。

> 写出正确的程序很难；写出正确的并发程序更难。与顺序程序相比，在并发程序中可能出错的事情更多。
>
> ——《Java 并发编程实践》，Brian Goetz 等著

因此，任何具有共享状态的工作负载都需要正确的保护和控制。对于在 JVM 上运行的工作负载来说，平台提供了一组名为 Java 内存模型（JMM）的内存保证。在深入介绍该模型之前，让我们先看一些解释 Java 并发问题的简单示例。

基本的Java并发

并发的反直觉性本质让我们认识到增量不是一个单独的操作，来看看下面这个示例：

```
public class Counter {
    private int i = 0;

    public int increment() {
        return i = i + 1;
    }
}
```

分析编译后的字节码可以发现一系列的指令，这些指令会导致加载、增量和存储值：

```
public int increment();
   Code:
   0: aload_0
   1: aload_0
   2: getfield      #2  // Field i:I
   5: iconst_1
   6: iadd
   7: dup_x1
   8: putfield      #2  // Field i:I
  11: ireturn
```

如果计数器没有受到适当的锁的保护，并且是以多线程的方式访问的，那么在另一个线程存储之前就有可能发生加载，从而导致更新丢失。

为了更详细地了解这个问题，可以考虑两个线程 A 和 B，让它们都在同一个对象上调用 `increment()` 方法。为了简单起见，假设它们都在一台单 CPU 的机器上运行，并且字节码准确地表示了底层的执行情况。因此，没有重新排序、缓存效应或实际处理器的其他细节。

> 由于操作系统调度器会在非决定性的时间导致线程的上下文切换，因此即使只有两个线程，也可以有很多不同的字节码序列。

假设单 CPU 执行的字节码如图所示（注意，这些指令的执行顺序是有明确定义的，在实

际的多处理器系统中是没有的）：

```
A0: aload_0
A1: aload_0
A2: getfield #2 // Field i:I
A5: iconst_1
A6: iadd
A7: dup_x1
B0: aload_0
B1: aload_0
B2: getfield #2 // Field i:I
B5: iconst_1
B6: iadd
B7: dup_x1
A8: putfield #2 // Field i:I
A11: ireturn
B8: putfield #2 // Field i:I
B11: ireturn
```

因为每个线程从它单独进入方法时就有一个私有的评价栈，所以只有字段上的操作才能相互干扰（因为对象字段位于堆中，且是共享的）。

由此产生的行为是，如果在 A 或 B 开始执行之前，`i` 的初始状态为 `7`，那么如果执行顺序恰好如刚才所示，则两个调用都会返回 `8`，而字段状态也会被更新为 `8`，尽管 `increment()` 被调用了两次。

> 这个问题是由操作系统的调度引起的，不需要任何硬件技巧就能发现它，而且即使是在一个没有现代特性的、非常老旧的 CPU 上也会出现这个问题。

另一个误解是添加关键字 `volatile` 会使增量运算安全。通过强迫该值总能被缓存重读，`volatile` 保证了任何更新都会被其他线程看到。但是它并不能解决刚才显示的更新丢失问题，因为这是增量运算符的复合性质造成的。

下面的例子显示了两个线程共享对同一个计数器的引用：

```java
package optjava;

public class CounterExample implements Runnable {

    private final Counter counter;

    public CounterExample(Counter counter) {
        this.counter = counter;

    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
                    + " Value: " + counter.increment());
```

```
            }
        }

    }
```

计数器没有 synchronized 或适当的锁的保护。当一个程序运行时，两个线程的执行可能会以不同的方式交错。代码在某些情况下会按照预期运行，而计数器会正常增量，这要归功于程序员的好运气！在另一些情况下，由于更新丢失，交错可能会在计数器中显示重复的值，就像下面看到的这样：

```
Thread-1 Value: 1
Thread-1 Value: 2
Thread-1 Value: 3
Thread-0 Value: 1
Thread-1 Value: 4
Thread-1 Value: 6
Thread-0 Value: 5
```

换句话说，一个在大多数时间内成功运行的并发程序和一个正确的并发程序是不一样的。证明它的失败和证明它是正确的一样困难，然而，只要找到一个失败的例子就足以证明它是不正确的。

更糟糕的是，重现并发代码中的错误非常困难。Dijkstra 有一句很经典的话："测试是检测错误的存在，而不是没有错误"，这句话相对单线程应用程序更适用于并发代码。

为了解决上述问题，可以使用 synchronized 来控制一个简单的值的更新，比如 int，在 Java 5 之前，这是唯一的选择。

使用同步的问题在于它需要一些精心的设计和前期的思考。如果不这样做，仅仅添加同步功能就会拖慢项目的速度，而不是加快它的速度。

这与添加并发的整个目的（即增加吞吐量）背道而驰。因此，任何对代码库进行并行化的练习都必须有性能测试来支持，以充分证明增加复杂性的好处。

> 添加同步块，特别是当不存在争用的情况下，成本要比旧版本的 JVM 低很多（但如果没有必要的话，仍然不应该这样做），更多细节参见 10.5.2 节。

为了更好地完成同步，你需要了解 JVM 的底层内存模型以及如何将其应用于并发应用程序的实际技术。

12.2　理解JMM

Java 从 1.0 版本开始就有了一个正式的内存模型，即 JMM。这个模型在 JSR 133[2] 中进行了大量修改，并修复了一些问题，它是作为 Java 5 的一部分交付的。

注 2：Java 平台是通过 Java 规范请求（Java Specification Request，JSR）来演进的。JSR 的作用是跟踪对平台标准的增强。

在 Java 语言规范中，JMM 是作为内存的数学描述出现的。它有着让人敬畏的名声，很多开发人员将其视为 Java 规范中最难攻克的部分（也许是除了泛型之外）。

JMM 试图提供诸如以下问题的答案：

- 当两个处理器核心访问相同的数据时会发生什么？
- 什么时候能保证它们看到的值是相同的？
- 内存缓存对这些答案有什么影响？

无论在任何地方访问共享状态，平台都会确保 JMM 所做的承诺得到遵守。这些承诺主要分为两大类：与排序相关的保证和与跨线程更新的可见性相关的保证。

随着硬件从单核系统到多核系统再到众核系统，内存模型的性质变得越来越重要。排序和线程可见性现在已经不再是理论问题，而是成为直接影响程序员日常代码的实际问题。

从上层看，像 JMM 这样的内存模型有两种可能的方案。

强内存模型（strong memory model）

所有核心在任何时候看到的值总是相同的。

弱内存模型（weak memory model）

不同的核心可能会看到不同的值，有特殊的缓存规则控制这种情况出现的时机。

从编程的角度来看，强内存模型看上去很有吸引力，特别是它不需要程序员在编写应用程序代码时小心翼翼。

图 12-3 显示了一个现代多处理器系统的简化视图，我们在第 3 章和第 7 章（在介绍 NUMA 架构的地方讨论过）曾见过这个视图。

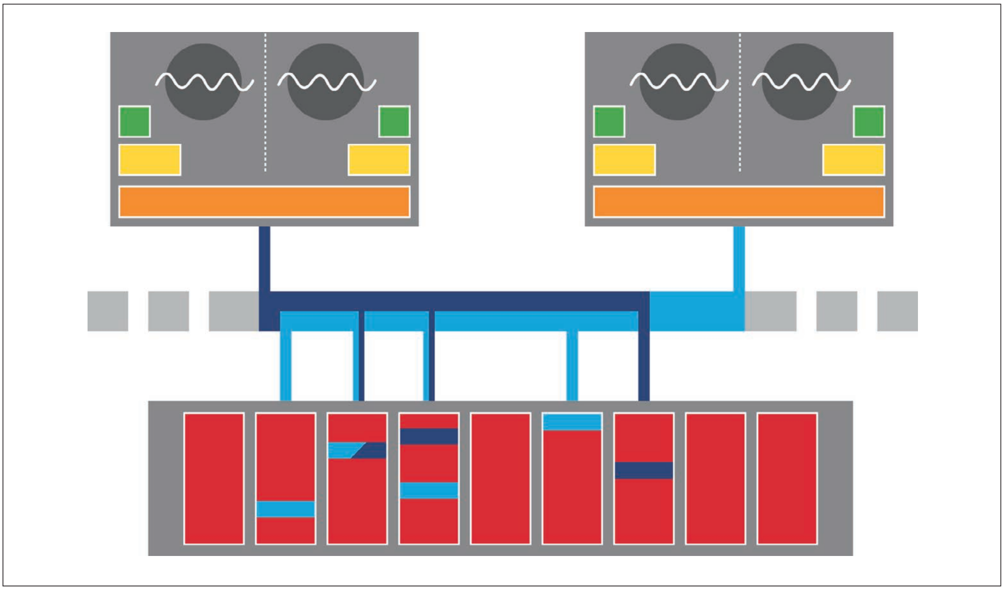


图 12-3：现代多处理器系统

如果在这种硬件上实现强内存模型，那就等同于写回内存的方法。缓存失效的通知将淹没内存总线，并且主存的有效传输率将急剧下降。随着核心数量的增加，问题只会越来越严重，从而使这种方法根本不适合众核领域。

另外值得记住的是，Java 被设计为一个独立于架构的环境。这意味着假如 JVM 指定的是强内存模型，如果要在不支持强内存模型的硬件上运行，就需要在软件层增加额外的实现工作，这进而会大大增加在弱内存模型硬件上实现 JVM 所需要的移植工作。

实际上，JMM 是一种非常弱的内存模型。这更符合真实 CPU 架构的发展趋势，包括 MESI（在 3.2.1 节介绍过）。因为 JMM 很少提供保证，所以这也使得移植变得更容易。

认识到 JMM 只是一个最低要求是非常重要的。真正的 JVM 实现和 CPU 可能比 JMM 要求的做得更多，我们在 3.3.3 节讨论过。

这可能会让开发人员得到一种错误的安全感。如果应用程序是在具有比 JMM 更强的内存模型的硬件平台上开发的，那么未发现的并发错误可以继续存在，因为有硬件保证它们在实际运行的时候不会表现出来。但是当同样的应用程序被部署到内存模型较弱的硬件上时，并发错误可能就会成为问题，因为该应用程序此时不再受到硬件的保护。

JMM 的保证是基于如下一组基本概念提供的。

先行发生

  一个事件确定在另一个事件之前发生。

同步

  事件将导致其对象的视图与主内存同步。

类串行

  指令在执行线程之外看上去是按顺序执行的。

先释放，再获取

  锁要先被某个线程释放，之后才可以被下一个线程获取。

处理共享可变状态的一个最重要的技术是通过同步进行锁定。它是 Java 并发性视图的一个基本部分，我们需要对它进行深入的讨论，以便充分地使用 JMM。

> 对于那些对性能感兴趣的开发人员来说，仅仅了解 `Thread` 类和 Java 并发机制的语言级基本原语是不够的。

在这个视图中，线程对对象的状态有自己的描述，它所做的任何改变都必须刷新到主内存中，然后由访问其他相同数据的线程重新读取。正如在 MESI 上下文中讨论的那样，这非常适合硬件的 write-behind 视图，但在 JVM 中，有相当多的实现代码包裹着底层内存访问。

从这个角度看，马上就能明白 Java 的关键字 `synchronized` 指的就是持有管程（monitor）的这个线程的本地视图已经与主内存同步（synchronized-with）。

同步的方法和块不仅定义了线程必须执行同步的接触点，还定义了一些代码块，这些代码块必须在其他同步的块或方法启动之前全部完成。

对于非同步访问，JMM 没有任何说明。在一个线程上所做的更改何时会对其他线程可见，它对此也没有任何保证。如果需要这样的保证，那么写访问必须用一个同步块来保护，这会触发缓存中的值被写入主内存。同样，读访问也必须包含在一个同步的代码段中以强制从内存中重新读取。

在现代的 Java 并发到来之前，使用 Java 的 `synchronized` 关键字是保证数据跨多个线程的排序和可见性的唯一机制。

JMM 会强制实施这种行为，并为 Java 和内存安全相关的假设提供各种保证。然而，传统的 Java 的 `synchronized` 锁有以下几个局限性，而且它们正变得越来越严重。

- 同等对待已锁定对象上的所有 `synchronized` 操作。
- 锁的获取和释放必须在一个方法的级别或一个方法内的 `synchronized` 块内完成。
- 要么获取锁，要么线程被阻塞；如果无法获取锁，则无法尝试获取锁并继续执行处理。

一个很常见的错误是忘记了必须公平对待已锁定数据上的操作。如果一个应用程序只是在写操作上使用 `synchronized`，那就会导致更新丢失。

比如，读操作似乎不需要锁定，但它必须使用 `synchronized` 来保证来自其他线程的更新的可见性。

> 线程之间的 Java 同步是一种合作机制，所以哪怕有一个参与的线程不遵守规则，它也不能正常工作。

对于 JMM 新手来说，可参阅 *JSR-133 Cookbook for Compiler Writers*。这本书简要解释了 JMM 的概念，没有包含太多会让读者应接不暇的内容细节。

比如，作为对内存模型的处理的一部分，很多抽象的屏障被引入进来并加以讨论。它们旨在允许 JVM 的实现者和库的作者以相对独立于 CPU 的方式思考 Java 并发的规则。

有关 JVM 实现实际上必须遵循的规则在 Java 语言规范中有详细的说明。在实践中，实现每种抽象屏障的实际指令在不同的 CPU 上可能不同。比如，Intel CPU 模型会在硬件中自动阻止某些重排序，所以上面提到的参考书中描述的一些屏障实际上不会执行任何操作。

最后要考虑的是：性能前景是一个不断变化的目标。自 JMM 诞生以来，无论是硬件的演进还是并发的前沿都没有停滞不前。因此，JMM 的描述并不能充分体现现代硬件和内存。

Java 9 也扩展了 JMM，试图赶上（至少部分赶上）现代系统的现状。其中一个关键的方面是它兼容其他编程环境，特别是 C++11，后者改编了来自 JMM 的一些理念，并进行了扩展。这意味着 C++11 模型提供了 Java 5 JMM（JSR 133）范围之外的概念定义。Java 9 对 JMM 进行了更新，将其中的一些概念带到了 Java 平台上，并支持底层的、和硬件相关的 Java 代码以一致的方式与 C++11 进行相互操作。

要想更深入地了解 JMM，可以参考 Aleksey Shipilëv 的博客文章“Close Encounters of the Java Memory Model Kind”，文中提供了很好的说明和详尽的技术信息。

12.3 构建并发库

尽管 JMM 设计得非常成功，但它很难理解，因此要转化为实际应用更是困难，与此相关的是内部锁也缺乏灵活性。

因此，自 Java 5 以来，不断增长的趋势是将高质量的并发库和工具作为 Java 类库的一部分进行标准化，并摆脱内置的语言级的支持。在绝大多数用例中，即使是那些对性能敏感的用例，这些库也比从头开始创建新的抽象更合适。

`java.util.concurrent` 中设计的库使得在 Java 中编写多线程更容易。Java 开发人员的工作就是选择最适合自己需求的抽象层次，而且幸运的是，从 `java.util.concurrent` 选择抽象恰当的库还能带来更好的“多线程”性能。

Java 提供的核心构建块可以分为以下几类：

- 锁（lock）和信号量（semaphore）
- 原子（atomic）
- 阻塞队列（blocking queue）
- 锁存器（latch）
- 执行器（executor）

图 12-4 显示了一个典型的现代并发 Java 应用程序的表示，它是基于并发原语和业务逻辑建立起来的。

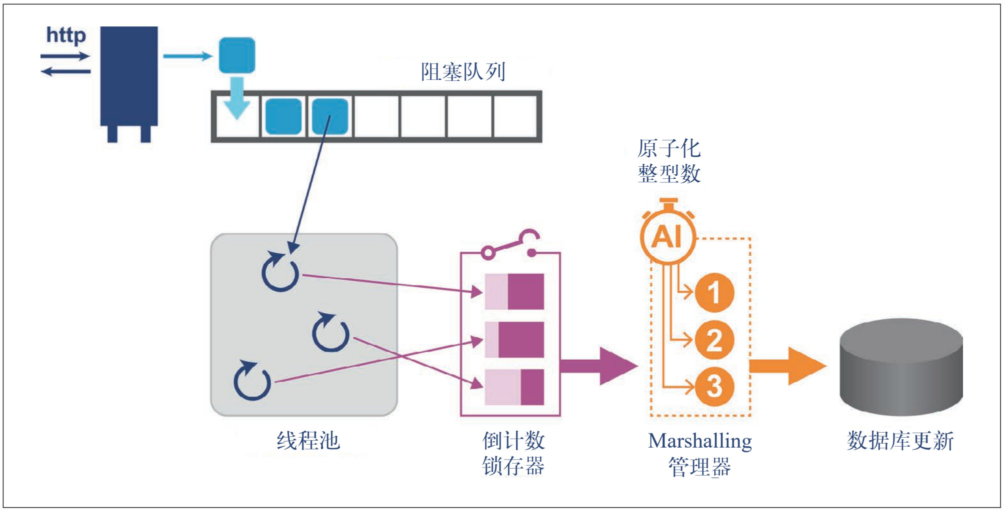


图 12-4：并发应用程序示例

其中的一些构建块将在下一节讨论，但是在学习它们之前，我们先来看看库中使用的一些

主要实现技术。了解并发库的实现方式将有助于注重性能的开发人员充分地利用它们。对于要处理极端情况的开发人员而言，如果标准库不能满足其团队的要求，那了解这些库的工作原理也能为他们选择（或开发）超高性能的替代品提供一个起点。

一般而言，这些库会尽量摆脱对操作系统的依赖，并且尽可能多地在用户空间工作。这有很多优点，其中最重要的一点是库的行为有望在全局范围内更加一致，而不是在不同的类似 Unix 操作系统之间还要受制于微小但重要的变化。

有些库（特别是锁和原子）要依靠底层的处理器指令和操作系统特性来实现一种叫作**比较和交换**（compare and swap，CAS）的技术。

这种技术需要一对值，即“预期的当前值”和“想要的新值”，以及一个内存位置（一个指针）。作为一个原子单元，会发生以下两个操作。

1. 将预期的当前值与位于指定内存位置的内容进行比较。
2. 如果匹配，则将当前值与想要的新值交换。

CAS 是用于一些更高级的关键并发特性的基本构建块，所以这也是一个典型的例子，说明自 JMM 产生以来性能和硬件的格局并没有停滞不前。

尽管在大多数现代处理器上，CAS 的特性是在硬件中实现的，但它并不是 JMM 或 Java 平台规范的一部分。相反，它必须被视为特定于实现的扩展，因此对 CAS 的硬件的访问是通过 `sun.misc.Unsafe` 类提供的。

12.3.1 Unsafe

`sun.misc.Unsafe` 是一个内部实现类，正如包名所示，它不是 Java 平台标准 API 的一部分。因此，应用程序开发人员一般不会直接使用它，而线索就在类名中。任何使用它的代码在技术上都会与 HotSpot 虚拟机直接产生耦合，这是潜在的风险。

> `Unsafe` 是一个不受支持的内部 API，因此从技术角度来讲，它可能随时被取消或修改，而不用考虑用户的应用程序。在 Java 9 中，它已经被放在了 `jdk.unsupported` 模块中，我们将在后面讨论。

然而，基本上每种主要框架的实现都以各种方式在关键的地方用到了 `Unsafe`，因为它提供了打破 JVM 标准行为的方法。比如，`Unsafe` 使如下操作成为可能：

- 分配一个对象，但不运行它的构造器；
- 访问原始内存和执行相当于指针运算的功能；
- 使用处理器特有的硬件特性（如 CAS）。

这些操作支持如下这些高级框架特性：

- 快速序列化 / 反序列化；
- 线程安全的原生内存访问（比如堆外内存或 64 位索引访问）；
- 原子内存操作；
- 高效对象 / 内存布局；

- 定制内存屏障；
- 和原生代码快速交互；
- JNI 的多操作系统替代品；
- 使用 volatile 语义访问数组元素。

虽然 `Unsafe` 并不是 Java SE 的官方标准，但它在业界的广泛使用已经使其成为事实上的标准。此外，它已经成为一些非标准但必要的特性的基础。不过，随着 Java 9 的到来，这种局面已经受到了影响，预计在接下来的几个 Java 版本中会有很大的变化。

接下来通过探索 Java 5 中引入的原子类来看看 CAS 这一技术的实际应用。

12.3.2　原子与CAS

原子类提供了一些复合操作来执行加、自增和自减法，它们都会结合一个 `get()` 来返回受到影响的结果，因此在两个独立的线程上执行自增操作将返回 `currentValue + 1` 和 `currentValue + 2`。原子变量的语义虽然是对 `volatile` 的扩展，但它们更加灵活，因为可以安全地执行状态相关的更新。

原子变量既不继承它们所封装的基础类型，也不允许直接替换。比如，`AtomicInteger` 类并没有扩展 `Integer` 类，一个原因是 `java.lang.Integer` 恰好是一个 `final` 类。

现在来看看 `Unsafe` 如何提供一个简单的原子调用的实现：

```
public class AtomicIntegerExample extends Number {

    private volatile int value;

    // 设置使用Unsafe.compareAndSwapInt来更新
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset(
                 AtomicIntegerExample.class.getDeclaredField("value"));
        } catch (Exception ex) {
            throw new Error(ex);
        }
    }

    public final int get() {
        return value;
    }

    public final void set(int newValue) {
        value = newValue;
    }

    public final int getAndSet(int newValue) {
        return unsafe.getAndSetInt(this, valueOffset, newValue);
    }
    // ...
```

这依赖于 `Unsafe` 的一些方法，而且这里的关键方法是原生的，会调用到 JVM 中：

```
public final int getAndSetInt(Object o, long offset, int newValue) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!compareAndSwapInt(o, offset, v, newValue));

    return v;
}

public native int getIntVolatile(Object o, long offset);

public final native boolean compareAndSwapInt(Object o, long offset,
                                int expected, int x);
```

该示例演示了在 `Unsafe` 内使用循环来反复重试 CAS 这一操作。要有效使用原子，至关重要的是开发人员要使用所提供的设施，而不是使用自己的实现，比如说在循环内使用原子自增操作，因为 `Unsafe` 实现在内部已经使用了该技术。

原子是无锁（lock-free）的，所以不会死锁。通常原子会伴随着一个内部重试循环来处理比较和更新失败的情况，这一般会发生在另一个线程刚刚执行更新时。

如果更新变量需要进行多次重试，那么此重试循环会导致性能线性下降。在考虑性能时，重要的是监控争用水平，以确保吞吐量保持在较高的水平。

使用 `Unsafe` 来提供对底层硬件指令的访问，这一点很有趣，因为 Java 通常支持开发人员与机器完全剥离。然而，在这种情况下，对机器指令的访问对于保证原子类的预期语义至关重要。

12.3.3　锁和自旋锁

到目前为止，我们所遇到的内部锁都是通过在用户代码中调用操作系统来工作的。该操作系统会让一个线程进入无限期的等待，直到收到信号。如果所争用的资源只使用很短的一段时间，那么这可能是一个巨大的开销。此时，让被阻塞的线程在 CPU 上保持活跃而不用完成任何有用的工作，然后重新尝试锁定，直到锁变为可用，这样来“燃烧 CPU”可能会更有效。

这种技术被称为**自旋锁**（spinlock），比完全的互斥锁更为轻量级。在现代系统中，自旋锁通常使用 CAS 来实现，并假设硬件支持 CAS。下面来看一个在底层 x86 汇编的简单示例：

```
locked:
     dd      0

spin_lock:
     mov     eax, 1
     xchg    eax, [locked]
     test    eax, eax
     jnz     spin_lock
     ret
```

```
spin_unlock:
     mov     eax, 0
     xchg    eax, [locked]
     ret
```

虽然自旋锁的具体实现因 CPU 而异，但其核心概念在所有系统上都是一样的。

- “测试并设置”（test and set）操作必须是原子的，这里通过 `xchg` 指令实现。
- 如果对自旋锁有争用，那么正在等待的处理器就会执行一个紧凑的循环。

如果预期的值是正确的，那么 CAS 本质上就会支持在一条指令中安全地更新一个值，进而帮助我们构造用于锁的构建块。

12.4 并发库总结

前面介绍了用于支持原子类和简单锁的底层实现技术，现在来看看标准库如何使用这些技术来创建通用的、功能齐全的生产库。

12.4.1 java.util.concurrent中的Lock

Java 5 重新设计了锁，并通过 `java.util.concurrent.locks.Lock` 为锁增加了一个更加通用的接口。与内部锁的行为相比，这个接口提供了更多的可能性。

`lock()`
: 以传统的方式来获取锁，会阻塞直到锁可用。

`newCondition()`
: 创建与这个锁有关的条件（condition），支持更灵活地使用该锁并支持分离锁内的关注点（比如读和写）。

`tryLock()`
: 尝试获取锁（可以指定超时时间），允许线程在锁不可用的情况下继续其处理。

`unlock()`
: 释放锁，这是跟在 `lock()` 后面的相应调用。

除了支持创建不同类型的锁之外，锁现在还可以跨多个方法，因为可以在一个方法中加锁，而在另一个方法中解锁。如果一个线程想以非阻塞方式获取锁，那可以使用 `tryLock()` 方法，这时如果锁不可用，它就会退回。

`ReentrantLock` 是 `Lock` 的主要实现，主要使用了一个带 `int` 的 `compareAndSwap()`。因此，在非争用的情况下，锁的获取是无锁的。这不仅可以极大地提升在锁争用较少时系统的性能，而且还提供了支持不同加锁策略的额外灵活性。

> 一个线程能够重新获取相同的锁，这种理念就是所谓的**可重入锁**，它可以防止线程把自己阻塞。大部分现代应用级别的锁机制是可重入的。

实际调用 compareAndSwap() 和使用 Unsafe 的地方可以在静态子类 Sync 中找到，而 Sync 继承自 AbstractQueuedSynchronizer。AbstractQueuedSynchronizer 还用到了 LockSupport 类，该类拥有支持挂起和恢复线程的方法。

LockSupport 类是通过向线程发放许可来工作的，如果没有有效的许可，线程就必须等待。这里许可的概念与信号量中发布许可的概念类似，但它只有一个许可（二进制信号量）。如果没有许可可用，线程就要挂起；而一旦有了有效的许可，那线程就可以从挂起恢复了。该类的方法取代了 Thread.suspend() 和 Thread.resume() 这两个早已被淘汰的方法。

有 3 种形式的 park() 方法会影响如下基本的伪代码。

```
while (!canProceed()) { ... LockSupport.park(this); }}
```

这 3 种形式的 park() 方法如下。

park(Object blocker)
: 阻塞，直到另一个线程调用 unpark()，线程会被中断或发生虚假唤醒。

parkNanos(Object blocker, long nanos)
: 行为与 park() 相同，但如果经过了指定的时间（以纳秒为单位），也会返回。

parkUntil(Object blocker, long deadline)
: 与 parkNanos() 类似，但是向场景中加入了一个会引发该方法返回的超时时间。

12.4.2 读/写锁

应用程序中很多组件的读操作和写操作的数量并不平衡。读操作不会改变状态，而写操作可以。使用传统的 synchronized 或 ReentrantLock（不用条件）会遵循单锁策略。在像缓存这样的情况下，可能会有很多个读者和一个写者，数据结构也可能会因为另一个读操作而花费大量的时间阻塞很多读者，而这是不必要的。

ReentrantReadWriteLock 类提供了一个 ReadLock 和一个 WriteLock，可以在代码中使用。其优势是多线程读操作不会导致其他线程阻塞，唯一会引发阻塞的操作是写。在读者数量较多的情况下，使用这种加锁模式可以极大提高线程的吞吐量并减少锁定。你也可以将锁设置为“公平模式”，虽然这样会降低性能，但可以保证线程按顺序处理。

与使用单锁的版本相比，AgeCache 的如下实现有很大改进：

```
package optjava.ch12;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class AgeCache {
        private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock readLock = rwl.readLock();
    private final Lock writeLock = rwl.writeLock();
    private Map<String, Integer> ageCache = new HashMap<>();
```

```
public Integer getAge(String name) {
    readLock.lock();
    try {
        return ageCache.get(name);
    } finally {
                    readLock.unlock();
            }
    }

    public void updateAge(String name, int newAge) {
     writeLock.lock();
     try {
        ageCache.put(name, newAge);
     } finally {
        writeLock.unlock();
     }
    }

}
```

然而，还可以通过考虑底层数据结构使其得到进一步优化。在这个示例中，并发集合是一种更合理的抽象，在多线程情况下效果更好。

12.4.3 信号量

信号量提供了一种独特的技术，支持访问多个可用的资源，比如池中的线程或数据库连接对象。信号量工作的前提是"最多有 X 个对象允许访问"，它通过一定数量的许可来控制访问。

```
// 支持两个许可的信号量，采用公平模式
private Semaphore poolPermits = new Semaphore(2, true);
```

`Semaphore::acquire()` 会将可用许可的数量减少一个，如果没有许可可用，则会阻塞。`Semaphore::release()` 会返回一个许可，如果有正在等待访问的，可以释放一个出来。因为信号量经常被用在资源可能会阻塞或排队访问的情况下，所以它很可能一开始就被初始化为公平模式，以避免线程饥饿。

只有一个许可的信号量（二进制信号量）等价于互斥量（mutex），但二者有一个明显的区别：互斥量只能由加锁的互斥量的线程来释放，而信号量可以由并不拥有它的线程来释放。像这样需要强制解决死锁的场景，二进制信号量可能就是必要的。信号量的另一个优势是它可以请求和释放多个许可。如果使用多个许可，就可以采用公平模式，否则会增加线程饥饿的机会。

12.4.4 并发集合

我们在第 11 章探索了可以对 Java 集合进行的优化。自 Java 5 以来，已经有专门为并发使用而设计的集合类接口的实现。随着时间的推移，这些并发集合也在修改和完善，以提供最佳的多线程性能。

比如，Map 实现（ConcurrentHashMap）使用了分成桶或段的方式，我们可以利用这种结构来实现性能的真正提升。每个段可以有自己的锁策略，也就是自己的一系列锁，同时拥有一个读锁和一个写锁来使多个读者横跨整个 ConcurrentHashMap 进行读取。如果需要写，则只需要在这一段上加锁。读者通常不用加锁就可以安全地与 put() 和 remove() 风格的操作重叠执行，他们还会观察到完整的更新操作遵循先行发生顺序。

需要注意的是，迭代器（以及用于并行流的分割迭代器）是作为一种快照获取的，因此它们不会抛出 ConcurrentModificationException。当碰撞过多时，表将被动态扩展，进而导致该操作的成本可能会非常高。如果你在编写代码的时候知道其大概的大小，那么提前给出来是值得的（和 HashMap 一样）。

Java 5 还引入了 CopyOnWriteArrayList 和 CopyOnWriteArraySet，它们在某些使用模式下可以提高多线程的性能。在使用这些类型的情况下，任何针对该数据结构的修改操作都会引发创建背后数组的一份新副本。任何现有的迭代器都可以继续遍历旧数组，一旦所有引用丢失，旧的数组副本就符合垃圾收集的条件了。同样，这种快照式的迭代方式可以确保不会引起 ConcurrentModificationException。

在读取操作比修改操作多很多的系统中使用写时复制（copy-on-write）数据结构，这种权衡表现良好。如果你正在考虑使用这种方法，那么记得在进行修改操作的同时使用一组良好的测试来测量性能改进。

12.4.5 锁存器和屏障

要控制一组线程执行，锁存器和屏障是很有用的技术。比如可以编写一个这样的系统，其中工作线程负责执行以下操作。

1. 从某个 API 中检索数据并进行解析。
2. 将结果写入数据库。
3. 最后，基于 SQL 查询计算结果。

如果系统只是简单地启动所有的线程运行，那么就无法保证事件的顺序。要是你希望让所有线程在启动任务 #3 之前，先完成任务 #1，再完成任务 #2，那一种可能的选择是使用**锁存器**。假设我们有 5 个线程在运行，代码可以这样写：

```
package optjava.ch12;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class LatchExample implements Runnable {

    private final CountDownLatch latch;

    public LatchExample(CountDownLatch latch) {
        this.latch = latch;
    }
```

```
    @Override
    public void run() {
        // 调用一个API
        System.out.println(Thread.currentThread().getName() + " Done API Call");
        try {
            latch.countDown();
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()
            + " Continue processing");
    }

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch apiLatch = new CountDownLatch(5);

        ExecutorService pool = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 5; i++) {
            pool.submit(new LatchExample(apiLatch));
        }
        System.out.println(Thread.currentThread().getName()
            +" about to await on main..");
        apiLatch.await();
        System.out.println(Thread.currentThread().getName()
            + " done awaiting on main..");
        pool.shutdown();
        try {
            pool.awaitTermination(5, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("API Processing Complete");
    }

}
```

在这个示例中，锁存器的计数被设置为 5，每个线程都会调用 `countdown()`，将计数减少 1。一旦计数达到 `0`，锁存器就会打开，任何在 `await()` 函数上等待的线程都会被释放，并继续进行处理。

需要注意的是，这种类型的锁存器是一次性的。一旦结果为 `0`，锁存器就不能重复使用，它是无法重置的。

> 锁存器在诸如启动期间的缓存填充和多线程测试等示例中非常有用。

在示例中，你可以使用两个不同的锁存器：一个用于等待 API 结果完成，另一个用于等待数据库结果完成。你也可以使用一个 `CyclicBarrier`，它可以被重置。然而，要弄清楚应该由哪个线程来控制重置也是相当有挑战的，它会涉及另一种类型的同步。一个常见的最佳实践是为流水线中的每个阶段使用一个屏障或锁存器。

12.5 执行器和任务抽象

在实践中，大多数 Java 程序员不应该处理底层的线程问题。相反，我们应该寻求使用 java.util.concurrent 中的一些特性，它们支持在适当的抽象层次上进行并发编程。比如，使用 java.util.concurrent 中的一些库，让线程保持忙碌，从而获得更好的多线程性能（比如让一个线程处于运行状态，而不是阻塞和处于等待状态）。

剥离了与线程相关的绝大部分重要内容的抽象层次可以描述为**并发任务**，它指的是我们需要在并发执行上下文内运行的代码或工作单元。将工作单元考虑为任务可以简化并发程序的编写，因为开发人员不必考虑用于执行任务的实际线程的生命周期。

12.5.1 认识异步执行

在 Java 中实现任务抽象的一种方式是使用 Callable 接口来表示会返回一个值的任务。Callable<V> 是泛型接口，其中定义了一个函数 call()，该函数会返回一个 V 类型的值，在无法计算结果的情况下会抛出异常。虽然表面上 Callable 与 Runnable 看起来非常相似，但是 Runnable 既不会返回结果，也不会抛出异常。

> 如果 Runnable 抛出了一个未被捕获的 Unchecked 异常，那它会在栈中传播，执行线程会默认停止运行。

在线程的生命周期内处理异常是一个很困难的编程问题，因为如果管理不当，就会导致 Java 程序以非正常状态继续运行。还应该注意的是，线程有可能被实现为操作系统级别的进程，从而导致在某些操作系统上创建线程的成本非常高昂。要从 Runnable 中获得任何结果，可能也会增加额外的复杂性，特别是要与另一个线程协调执行返回时。

Callable<V> 类型为我们提供了一种很好地处理任务抽象的方法，但是这些任务实际是如何执行的呢？

ExecutorService 是一个接口，它定义了在一个托管的线程池上执行任务的机制，而它的实际实现定义了池中的线程应该如何管理以及应该有多少个线程。ExecutorService 可以通过 submit() 方法及其重载版本来接收 Runnable 或 Callable。

辅助类 Executors 有一系列 new* 工厂方法，这些方法可以根据所选择的行为来构造服务和底层的线程池。通常使用以下这些工厂方法来创建新的执行器对象。

newFixedThreadPool(int nThreads)

　　构造一个具有固定大小的线程池的 ExecutorService，其中的线程将被重复使用以运行多个任务，从而避免了为每个任务多次创建线程的成本。当所有的线程都在使用时，新的任务会被保存在一个队列中。

newCachedThreadPool()

　　构造一个 ExecutorService，它将根据需要创建新的线程，并尽可能重复使用线程。已创建的线程会被保留 60 秒，之后会从缓存中删除。对于小型的异步任务，使用这个线

程池可以提供更好的性能。

`newSingleThreadExecutor()`

构造一个底层只有单个线程的 ExecutorService。任何新提交的任务都会被排队，直到该线程可用。这种类型的执行器可以用于控制并发执行的任务数量。

`newScheduledThreadPool(int corePoolSize)`

它创建的 ExecutorService 拥有一系列额外的方法，可以接受 Callable 类型和一个延迟参数，让任务在未来的某个点执行。

一旦任务被提交，它将被异步处理，用于提交任务的代码可以选择以阻塞或轮询的方式等待结果。调用 ExecutorService 的 submit() 方法会返回一个 Future<V>，它支持阻塞式的调用——get() 或指定超时时间的 get()，或非阻塞式的调用——以通常方式使用 isDone()。

12.5.2 选择一个ExecutorService

选择恰当的 ExecutorService 可以很好地控制异步处理，而且如果为线程池选择了合适的线程数，就可以产生明显的性能收益。

我们也可以编写一个自定义的 ExecutorService，但是一般没有必要，原因之一是类库提供了定制选项 ThreadFactory。ThreadFactory 允许类的作者编写定制的线程创建代码，也可以设置线程的属性，比如名称、守护状态和线程优先级。

在整个应用程序的设置中，ExecutorService 有时也需要根据经验进行调优。充分了解服务运行将要依赖的硬件和其他竞争资源是调优工作中很有价值的一部分。

通常要用到的一个指标是核心的数量与池中线程数量的对比情况。如果所选的要并发运行的线程数量高于可用的处理器数量，那就可能会出现问题并引发争用。操作系统需要调度要运行的线程，而这又会引发上下文切换。

当争用达到特定阈值时，转向并发处理所获得的性能优势就会往相反方向发展了。这就是为什么对于一个好的性能模型，能够测量性能改进（或损失）是必不可少的。我们在第 5 章讨论了性能测试技术和进行性能测试时应该避免的反模式。

12.5.3 Fork/Join

Java 提供了几种不同的实现并发的方法，不需要开发人员控制和管理自己的线程。Java 7 引入了 Fork/Join 框架，提供了一个新的 API，可以有效地配合多个处理器工作。它基于 ExecutorService 的一个叫作 ForkJoinPool 的新实现。这个类提供了一个托管的线程池，它有以下两个特性：

- 它可以用于高效地处理细分任务；
- 它实现了**工作窃取**（work-stealing）算法。

对细分任务的支持是通过 ForkJoinTask 类引入的，这是一个类似线程的实体，但是比标准的 Java 线程更为轻量级。预期的使用场景是，潜在的大量任务和子任务可以由 ForkJoinPool 执行器中的少量实际线程来支撑。

`ForkJoinTask` 的关键是它可以将自己反复细分为“更小”的任务，直到任务规模小到可以直接计算。为此，该框架只适合特定类型的任务，比如计算纯函数或其他“自然并行”的任务。即使是那样，要充分利用 Fork/Join 的这部分功能，可能也需要重写算法或代码。

然而，Fork/Join 框架的工作窃取算法部分可以独立于任务细分使用。比如，如果一个线程已经完成了分配给它的所有工作，而另一个线程有积压工作，那它将从繁忙线程的队列中窃取工作。这种在多个线程之间重新平衡作业的想法非常简单，但又很聪明，带来的效果也很可观。在图 12-5 中可以看到工作窃取的演示。

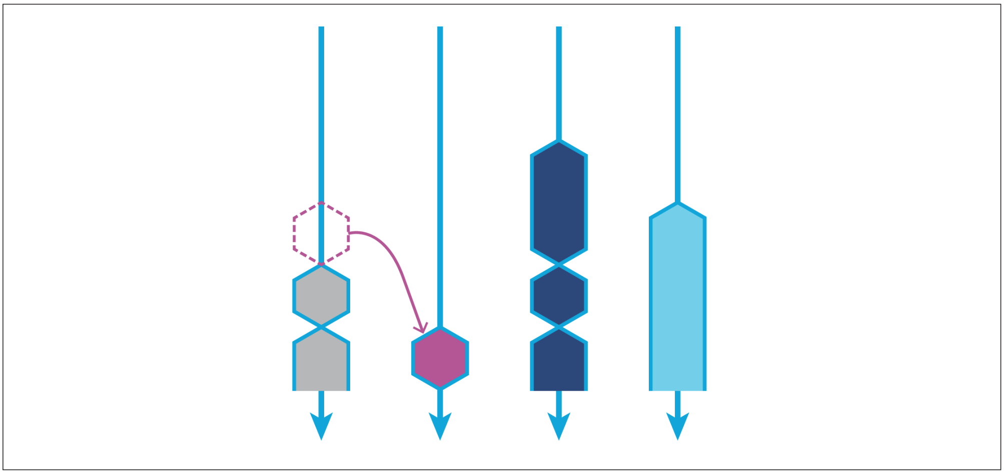


图 12-5：工作窃取算法

`ForkJoinPool` 有一个静态方法 `commonPool()`，它会返回指向系统级的池的引用。这不仅能让开发人员不必创建自己的池，还为共享提供了机会。这个公共的池采用的是惰性初始化策略，所以只在需要的时候才会创建。

这个池的大小被定义为 `Runtime.getRuntime().availableProcessors() -1`。然而，这个方法未必总能返回预期的结果。

Heinz Kabutz 在 Java Specialists 邮件列表上写道，他发现了一个案例，在一台配备了 16 个插槽，每个插槽有 4 个核心，每个核心有 2 个超线程的机器上，返回值为 16，这看上去是非常低的。根据在计算机上测试所获得的直觉，我们期望的结果是 16 * 4 * 2 = 128。然而，如果在这台机器上运行 Java 8，那它会将公共的 Fork/Join 池的并行度设置为只有 15。

> 虚拟机并不是真的知道处理器是什么，它只是向操作系统询问一个数字。同理，操作系统通常也不会在意，它会询问硬件。硬件会回应一个数字，通常是“硬件线程”的数量。操作系统相信硬件，虚拟机相信操作系统。
>
> ——Brian Goetz

幸运的是，有一个标志允许开发人员以可编程的方式设置预期的并行度：

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=128
```

不过正如第 5 章中所讨论的，要小心使用魔法标志。而且正如我们将讨论的选择 `parallelStream` 选项一样，没有什么是免费的！

Fork/Join 的工作窃取正越来越多地被库和框架开发人员所利用，即便没有任务细分。比如将在 12.6.3 节介绍的 Akka 框架，就是为了使用工作窃取而用到了 `ForkJoinPool`。Java 8 的到来也大大提升了 Fork/Join 的使用水平，因为 `parallelStream()` 在背后使用了公共的 Fork/Join 池。

12.6　现代Java并发

Java 中的并发最初是为这样的环境设计的：有些会长期运行但存在阻塞的任务，比如 I/O 和其他类似的缓慢操作，等待的时间就可以供其他线程交错执行。现在，几乎开发人员为其编写代码的每一台机器都是多处理器系统，所以高效利用可用的 CPU 资源是非常明智的。

然而，在 Java 将并发内置其中时，业界对此并没有很多经验。事实上，Java 是第一个在语言级别内置了线程支持的工业级标准环境。因此，开发人员在并发方面的许多惨痛教训是在 Java 中首次遇到的。Java 一般不会废弃特性（尤其是核心特性），所以 `Thread` API 仍然是 Java 的一部分，而且永远都是。

这就导致了这样一种情况，即在现代应用程序的开发中，与 Java 程序员习惯于编写的代码的抽象层次相比，线程是相当底层的。比如，在 Java 中我们不需要手动管理内存，那么为什么 Java 程序员要处理线程创建和其他生命周期事件等底层工作呢？

幸运的是，现代 Java 提供了一个环境，使我们可以利用内置于语言和标准库中的抽象来获得显著的性能提升。因此，开发人员一方面可以利用并发编程的优势，另一方面减少了面对底层编程时的挫折和样板代码。

12.6.1　流和并行流

到目前为止，Java 8 的最大变化（可以说是有史以来最大的变化）是引入了 Lambda 和流。将两者一起使用可以提供一种“神奇的开关”，让 Java 开发人员能够获得函数式编程风格的一些好处。

抛开 Java 8 作为一门语言实际能达到的函数式程度这个相当复杂的问题不谈，我们可以认为 Java 现在有了一种新的编程范式。这种更加函数式的风格关注的是数据，而不是它一直使用的、命令式的以及面向对象的方法。

Java 中的流是一个不可变的数据项序列，传达来自数据源的元素，它可以是来自任何类型的数据源（集合与 I/O）。我们在流上使用诸如 `map()` 等接收 Lambda 表达式或函数对象的操纵操作来处理数据。从外部迭代（传统的 `for` 循环）到内部迭代（流）的这种转变为数据的并行化和复杂表达式的惰性求值带来了良好的机会。

现在所有的集合都提供了来自 `Collection` 接口的 `stream()` 方法。这是一个默认方法，它提供了一个从任何集合创建流的实现，并在背后创建了一个 `ReferencePipeline`。

第二个方法是 `parallelStream()`，可以用来并行地处理数据项，并重新组合结果。使用 `parallelStream()` 会涉及使用一个 `Spliterator` 来对工作进行分解，并在公共的 Fork/Join 池上执行计算。这是处理自然并行问题的一种很方便的技术，因为流的项是不可变的，所以就避免了在并行处理时修改状态的问题。

与使用 `RecursiveAction` 重新编码相比，流的引入带来了一种在语法上更友好地使用 Fork/Join 的方式。用数据来表达问题与任务抽象类似，开发人员不必考虑底层的线程机制和数据易变性的问题。

总是使用 `parallelStream()` 可能很吸引人，但是使用这种方法也是有成本的。和任何并行计算一样，工作必须被分解为可以跨多个线程的任务，然后重新组合结果，这是 Amdahl 定律的一个直接的例子。

在较小的集合上，串行计算实际上可能更快。在使用 `parallelStream()` 时，务必总是保持谨慎，并进行性能测试。不能充分利用计算能力可能损失惨重，但是“测量，而不是猜测”在这里也适用。在使用并行流提升性能时，收益应该是直接、可测量的，所以不要盲目地将顺序流转换为并行。

12.6.2 无锁技术

无锁技术的出发点是阻塞对吞吐量不利，会降低性能。阻塞的问题是它相当于向操作系统表明有机会对当前线程进行上下文切换。

考虑一个在双核机器上运行 t1 和 t2 两个线程的应用程序。加锁可能会导致线程被上下文切换出去，之后回到另一个处理器上。此外，暂停和唤醒所需的时间会很显著，从而导致加锁技术可能比无锁技术慢很多。

Disruptor 模式可以突显无锁并发的潜在性能收益，它是一种现代设计模式，最初由 LMAX（London Multi Asset Exchange）引入。当与 `ArrayBlockingQueue` 进行基准测试对比时，Disruptor 的表现高出了几个数量级。GitHub 项目页面列出了其中的一些对比，我们从该页面选择了一个示例，性能数字如表 12-1 所示。

表12-1：LMAX性能统计：以每秒操作数表示的吞吐量

	`ArrayBlockingQueue`	Disruptor
Unicast: 1P–1C	5 339 256	25 998 336
Pipeline: 1P–3C	2 128 918	16 806 157
Sequencer: 3P–1C	5 539 531	13 403 268
Multicast: 1P–3C	1 077 384	9 377 871

这些惊人的结果是通过自旋锁实现的。两个线程之间的同步实际上是通过一个 `volatile` 变量（以确保跨线程的可见性）来手动控制的：

```
private volatile int proceedValue;

//...

while (i != proceedValue) {
```

```
        // 忙循环
    }
```

保持 CPU 核心旋转，意味着一旦接收到数据就可以立即在该核心上操作，而不需要进行上下文切换。

当然无锁技术也是有成本的。一直占用着一个 CPU 核心会导致在利用率和功耗方面成本高昂，因为计算机越来越忙，但实际上什么也没有做，而越忙就会越热，从而需要更多功耗来给什么都没处理的核心降温。

运行需要这种吞吐量的应用程序，往往要求程序员对软件的底层机制有很好的理解。这里又可以引用机械共鸣（代码如何与硬件交互）这个方法作为补充。这并非巧合，**机械共鸣**（mechanical sympathy）这个术语正是由 Martin Thompson 引入的，而他也是 Disruptor 模式的提出者之一。

> “机械共鸣”这个名字来自伟大的赛车手 Jackie Stewart，他曾 3 次获得国际汽车联合会世界一级方程式锦标赛冠军。他相信最佳车手对机器如何工作有足够的理解，所以他们能与赛车协调一致。
>
> ——Martin Thompson

12.6.3　基于Actor的技术

近年来，出现了几种不同的方法来表示任务，在某种程度上它们比线程**小一些**，我们已经在 `ForkJoinTask` 类中见过这种理念。另一种流行的方法是 **Actor 范型**（actor paradigm）。

Actor 是小型的、自包含的处理单元，它包含自己的状态，有自己的行为，还包含一个用来与其他 Actor 通信的邮箱（Mailbox）系统。Actor 解决状态问题的思路是不共享任何可变状态，只通过不可变的消息相互通信。Actor 之间的通信是异步的，它会响应收到的消息，执行指定任务。

在一个并行系统中，每个 Actor 都有特定的任务，它们构建了一个网络，这样就从底层的并发模型中完全剥离出来了。

Actor 可以处于同一个进程之内，不过这并不是必需的。这样做的好处是 Actor 系统可以是多进程的，甚至可以跨多台机器。当需要一定程度的容错能力时，多机和集群使得基于 Actor 的系统能够高效执行。为了确保 Actor 在协作环境中成功工作，它们通常有一个快速失败（fail-fast）策略。

对于基于 JVM 的语言，Akka 是用于开发基于 Actor 的系统的一个流行框架。该框架是用 Scala 编写的，但是也提供了 Java API，使其可以用于 Java 和其他 JVM 语言。

Akka 和基于 Actor 的系统的动机源于使并发编程变得困难的几个问题。Akka 文档强调了考虑使用 Akka 而不是传统加锁机制的 3 个核心动机：

- 在领域模型内封装可变状态非常棘手，尤其是当指向对象内部的引用可以在不受控制的情况下逃逸时；
- 用锁来保护状态会导致吞吐量明显降低；

- 锁可能会导致死锁和其他类型的活跃性问题。

其他突出的问题包括难以正确地使用共享内存，以及强迫多个 CPU 共享高速缓存运行而带来的性能问题。

最后要讨论的一个动机与传统线程模型和调用栈中的故障有关。底层的线程 API 没有处理线程失败或恢复的标准方法，而 Akka 对此进行了标准化，并为开发人员提供了一个定义良好的恢复机制。

总之，Actor 模型是开发人员的并发工具箱的一个很有用的补充。然而，它并不是所有其他技术的通用替代品。如果用例符合 Actor 风格（异步传递不可变的消息，没有共享的可变状态，每个消息处理器的执行都有时间限制），那么使用 Actor 就可以快速见效。但是，如果系统设计包括请求 – 响应的同步处理、共享可变状态或无约束执行，那么细心的开发人员可能会选择使用另一种抽象来构建自己的系统。

12.7　小结

对于希望使用多线程来改进应用程序性能之前应该考虑哪些主题，本章只触及了一些表面内容。在将单线程应用程序转变为并发设计的时候，应该：

- 确保能够准确测量线性处理的性能；
- 应用一个变化，并测试性能是否真的得到了提高；
- 确保性能测试易于重新运行，特别是当系统处理的数据大小可能发生变化时。

应该避免以下诱惑：

- 到处使用并行流；
- 利用手动锁定创建复杂的数据结构；
- 重新发明 `java.util.concurrent` 中已经提供的结构。

应该致力于：

- 使用并发集合以改进多线程性能；
- 使用利用了底层数据结构的访问设计；
- 减少跨整个应用程序的锁；
- 提供恰当的任务 / 异步抽象，避免手动处理线程。

退一步说，并发是未来高性能代码的关键。然而：

- 共享可变状态非常困难；
- 要正确使用锁很有挑战；
- 同步和异步的状态共享模型都是必需的；
- JMM 是非常底层、灵活的模型；
- 线程抽象是非常底层的。

现代并发的趋势是向更高层次的并发模型发展，脱离越来越像“并发的汇编语言”的线程。最近的 Java 版本增加了程序员可用的、更高级别的类和库。总之，业界似乎正在转向一种这样的并发模型：为了安全的并发抽象，更多的责任交由运行时和库来管理。

第 13 章

剖析

在程序员群体中，**剖析**（profiling）这个术语的使用并不是非常统一。事实上，可能的剖析方法有很多种，其中最常见的有以下两种：

- 执行
- 分配

本章将涵盖这两个主题。首先重点关注执行剖析，我们会借着这个主题来介绍可用于剖析程序的工具。之后会介绍内存剖析，看一看各种工具是如何提供这种能力的。

我们将探讨对于 Java 开发人员和性能工程师而言，了解剖析器的一般操作方式是多么重要。因为剖析器有可能扭曲应用程序的行为，并表现出明显的偏差。

执行剖析是性能剖析的领域之一，在这个领域中，这些偏差就会凸显出来。谨慎的性能工程师会意识到这种可能性，并通过各种方式来弥补，包括使用多种工具进行剖析，以了解真正发生的情况。

对于性能工程师而言，同样重要的是要解决自己的认知偏差，不要致力于挖掘符合自己预期的性能行为。在第 4 章中遇到的反模式和认知陷阱就是我们训练自己避免这些问题的一个很好的开始。

13.1　认识剖析

一般而言，JVM 剖析和监控工具的工作原理是，使用某个底层的注入（instrumentation）机制，将数据发送给图形控制台，或将其保存在日志中供后续分析。而底层的注入机制，通常采取的形式又不外乎是代理——在程序启动时加载，或者是组件——动态连接到运行中的 JVM。

我们在 2.7 节曾介绍过代理。它们是一种非常通用的技术，广泛应用于 Java 的各种工具中。

从广义上说，我们需要区分**监控工具**（其主要目标是观测系统及其当前状态）、**警报系统**（用于检测异常或反常行为）和**剖析器**（提供关于运行中应用程序的深层信息）。虽然这些工具目标不同，但是它们通常是相关的。一个运行良好的生产应用程序可能会用到所有这些工具。

然而，本章的重点是剖析。（执行）剖析的目的是找出用户编写的代码中可以进行重构和性能优化的部分。

剖析通常是通过向正在执行应用的 JVM 上连接一个自定义代理来实现的。

正如 3.6 节所讨论的，诊断和纠正性能问题的第一步是找出导致出现问题的资源。这一步如果定位错了，就会付出高昂的成本。

如果对一个 CPU 资源并没有耗尽的应用程序进行剖析，那么我们很容易被剖析器的输出严重误导。我们在 5.2 节曾引用 Brian Goetz 的话，这里可以拓展一下，即工具总是会给出一个数，只是不清楚这个数与所要解决的问题是否相关。正因为如此，我们在 4.5 节曾介绍了主要的偏差类型，并将有关剖析技术的讨论推迟到现在。

> 一个好的程序员……仔细观察关键代码是明智之举，但首先要找到关键代码。
>
> ——Donald Knuth

这意味着在进行剖析工作之前，性能工程师应该已经找到了性能问题。不仅如此，他们还应该证明确实是应用程序代码的问题。如果应用程序在用户模式下的 CPU 利用率接近 100%，那他们就会知道是这种情况。

如果不符合这些标准，那性能工程师应该从其他地方寻找问题的根源，而不是试图借助执行分析器做进一步诊断。

即使 CPU 在用户模式下（不是内核时间）完全耗尽，在进行剖析之前也必须排除另一个可能的原因，即垃圾收集的 STW 阶段。因为所有重视性能的应用程序都应该通过日志记录垃圾收集事件，所以这个检查很简单，也就是查阅该机器的垃圾收集日志和应用程序日志，确保垃圾收集日志很安静，而应用程序日志在活动。如果垃圾收集日志是在活动的那个，那么下一步应该是先对垃圾收集进行调优，而不是直接做执行剖析。

13.2 采样与安全点偏差

执行剖析的一个关键方面是它通常使用采样来获取正在运行的代码的数据点（栈轨迹）。毕竟，测量也是有成本的，所以为了防止数据收集成本过高，通常不会跟踪所有方法的进

入和退出。相反，我们会记录线程执行的快照，但这也只能以相对较低的频率进行，以免产生过高的开销。

比如，New Relic 线程剖析器（New Relic 技术栈提供的工具之一）是每 100 毫秒采样一次。这个限制通常被认为是在不产生过高开销的情况下，采样频率的最佳估计值。

采样间隔代表了性能工程师的一个权衡。采样太频繁，开销就会变得过高，特别是对于一个性能非常重要的应用程序来说。相反，如果采样频率太低，那错失重要行为的机会就会很大，进而可能导致采样无法反映应用程序的真实性能。

> 在使用剖析器时，它应该获得尽可能多的细节，以免得到意外的结果。
>
> ——Kirk Pepperdine

采样不仅使得问题有可能被隐藏在数据中，而且在大多数情况下，它只在安全点执行。这就是所谓的**安全点偏差**，其会导致以下两个后果。

- 所有的线程都必须到达某个安全点才能被采样。
- 样本只能是应用程序处于某个安全点的状态。

其中第一个后果是从正在运行的进程中生成剖析样本会带来额外的开销。第二个后果是我们知道采样的是处于某个安全点的状态，只采样这样的点会扭曲样本点的分布。

大多数执行剖析器使用来自 HotSpot 的 C++ API 中的 `GetCallTrace()` 函数来收集每个应用程序线程的栈样本。通常的设计是在代理内收集样本，然后记录数据或执行其他下游处理。

然而，`GetCallTrace()` 的开销相当大。因为如果有 *N* 个活动的应用程序线程，那么收集一个栈样本将导致 JVM 进入安全点 *N* 次。这个开销也是设置采样频率上限的一个根本原因。

因此，细心的性能工程师需要留意应用程序在安全点上花费的时间。如果花费时间太多，那应用程序的性能就会受到影响，进而导致任何调优工作都可能是在不准确的数据上进行的。我们可以用一个 JVM 标志来跟踪安全点时间过长的情况：

```
-XX:+PrintGCApplicationStoppedTime
```

这将把与安全点时间相关的额外信息写入垃圾收集日志中。一些工具（特别是 jClarity Censum）可以从这个标志产生的数据中自动检测出问题。Censum 还可以区分安全点时间和操作系统内核导致的暂停时间。

我们可以用一个例子，即一个**计数循环**来说明由安全点偏差造成的问题。这是一个简单的循环，类似于以下的代码段：

```
for (int i = 0; i < LIMIT; i++) {
    // 循环体中只有“简单”操作
}
```

在这个例子中，我们特意撇开了“简单”操作的含义，因为其行为取决于 JIT 编译器所能执行的具体优化，更多相关细节参见 10.4 节。

简单操作的例子包括基本类型的算术运算和已经完全内联的方法调用（在循环体内实际上已经没有方法了）。

如果 LIMIT 很大，那么 JIT 编译器会将该 Java 代码直接翻译为等价的编译形式，包括一个向后的分支回到循环头部。正如 10.9 节所讨论的，JIT 编译器会在循环的回边上插入安全点检查。这意味着对于一个很大的循环，每次循环迭代都有一次进入安全点的机会。

然而如果 LIMIT 足够小，那这种情况就不会发生，相反 JIT 编译器会展开循环。这意味着对于执行次数足够小的计数循环，线程直到循环结束后才会进入安全点。

因此，只在安全点采样会直接导致这样一种行为偏差：结果对于循环的大小和循环中执行的操作的性质非常敏感。

显然这不利于得到严谨可靠的性能结果。这也不是一个理论上的问题——循环展开可能会产生大量的代码，从而导致长长的代码块永远不会被采样到。

我们还将回到安全点偏差这个问题上，它仍然是证明性能工程师需要意识到各种权衡的一个很好的例子。

13.3 面向开发人员的执行剖析工具

本节将讨论几种不同带有图形用户界面的执行剖析工具。市面上这类工具有很多，所以我们重点选择一些最常见的进行讨论，不做详尽调查。

13.3.1 VisualVM剖析器

让我们将 VisualVM 作为第一个剖析工具的例子。它包括执行剖析器和内存剖析器，非常简单而且免费。不过它的局限性也很大，很少作为生产工具使用，但对于想了解其应用程序在开发和测试环境中行为的性能工程师来说，它又是很有帮助的。

图 13-1 显示了 VisualVM 的执行剖析视图。

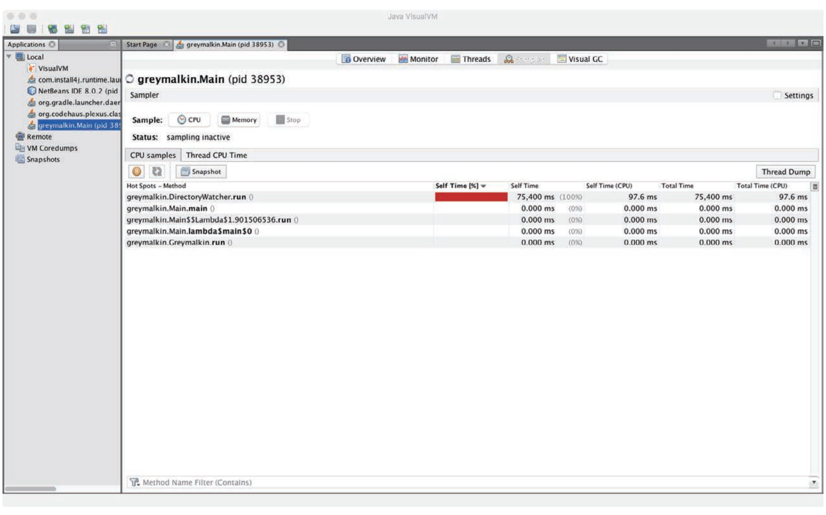


图 13-1：VisualVM 内存剖析器

这里演示了执行方法及相关 CPU 消耗情况的一个简单视图。我们能在 VisualVM 中挖掘的东西相当有限。因此，大多数性能工程师很快就会不满足于它了，转而使用市面上某个更完整的工具。然而，对于刚接触剖析艺术与权衡的性能工程师而言，它可能是很有用的第一个工具。

13.3.2 JProfiler

一个流行的商用编译器是 ej-technologies 有限责任公司开发的 JProfiler。这是一个基于代理的剖析器，用于剖析本地或远程的应用程序，既能以 GUI 工具的形式运行，也能以 Headless 模式运行。它兼容相当广泛的操作系统，除了常见的 Windows、macOS 和 Linux，还兼容 FreeBSD、Solaris 和 AIX 等系统。

这个桌面应用程序在第一次启动时会显示如图 13-2 所示的界面。

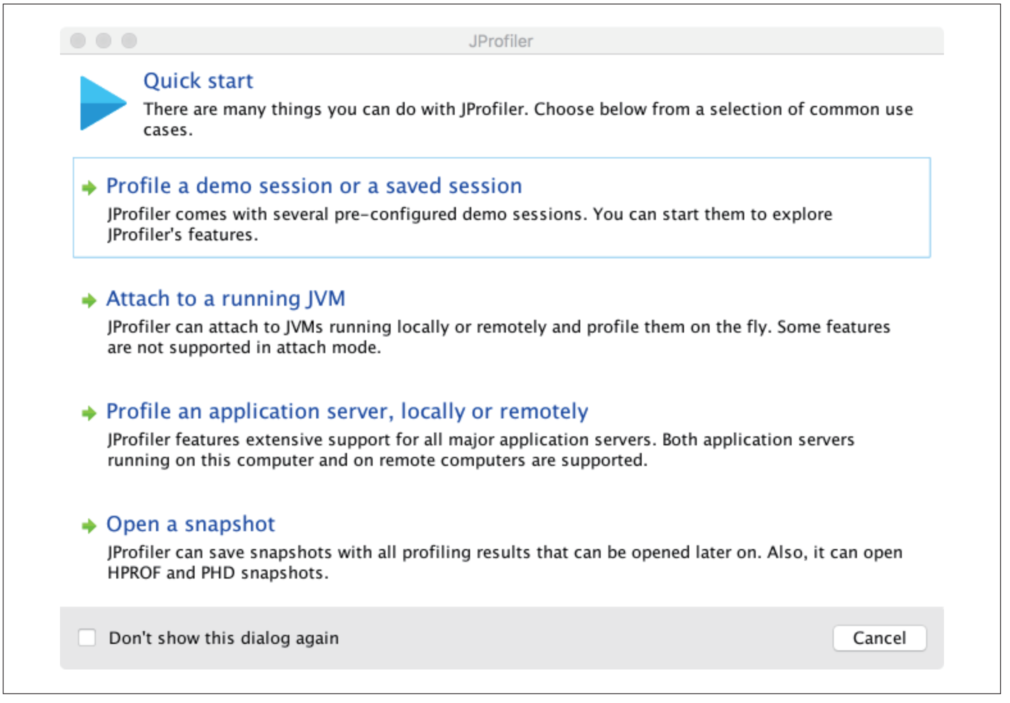


图 13-2：JProfiler 启动向导

关闭该界面后，可以看到如图 13-3 所示的默认界面。

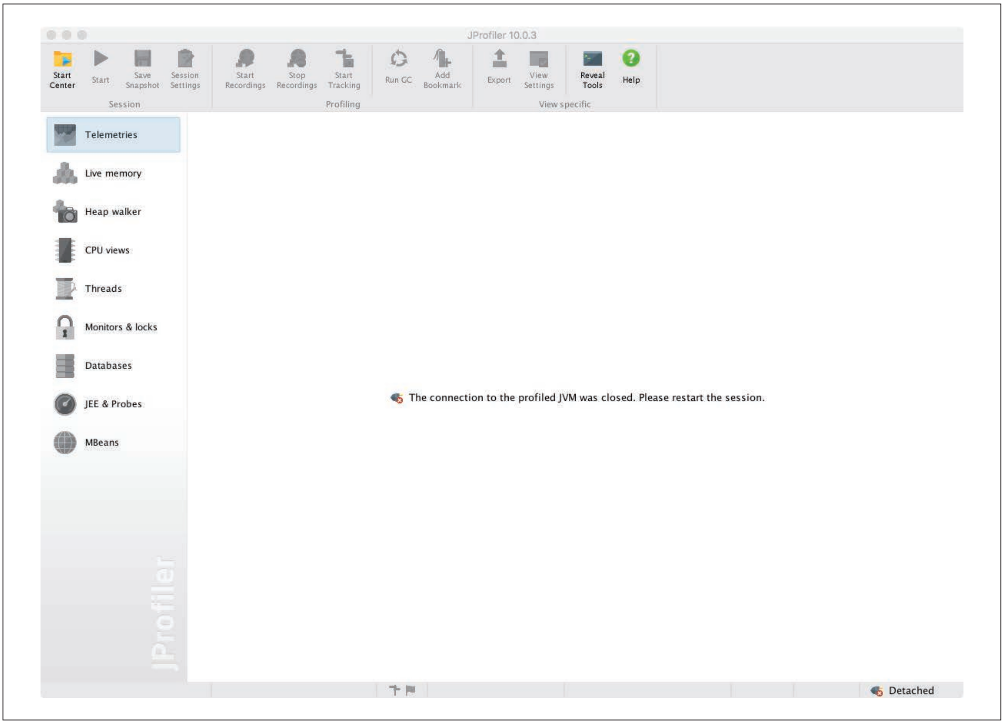


图 13-3：JProfiler 启动界面

点击左上角的启动中心（Start Center）按钮，可以看到很多选项，包括打开会话（Open Session），这里包含一些预先封装好的示例供用户使用，接下来就是快速附加（Quick Attach）。图 13-4 演示了快速附加选项，这里选择剖析 AsciidocFX 应用程序，它是一个创作工具，本书的很多内容就是用该工具编写的。

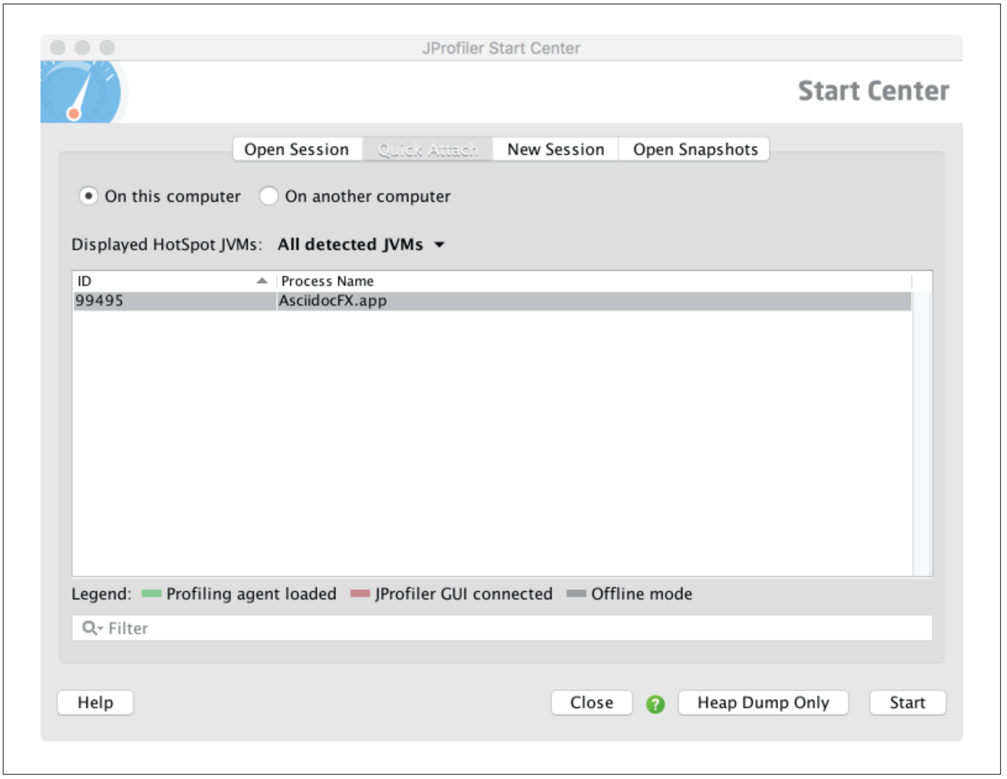


图 13-4：JProfiler 快速附加界面

附加到目标 JVM 会打开一个配置对话框，如图 13-5 所示。请注意，这款剖析器在工具的配置中已经就性能权衡、需要的有效过滤器和考虑剖析的开销给出了警告。正如本章前面所讨论的，执行剖析在很大程度上并不是灵丹妙药，工程师必须小心行事以避免混淆。

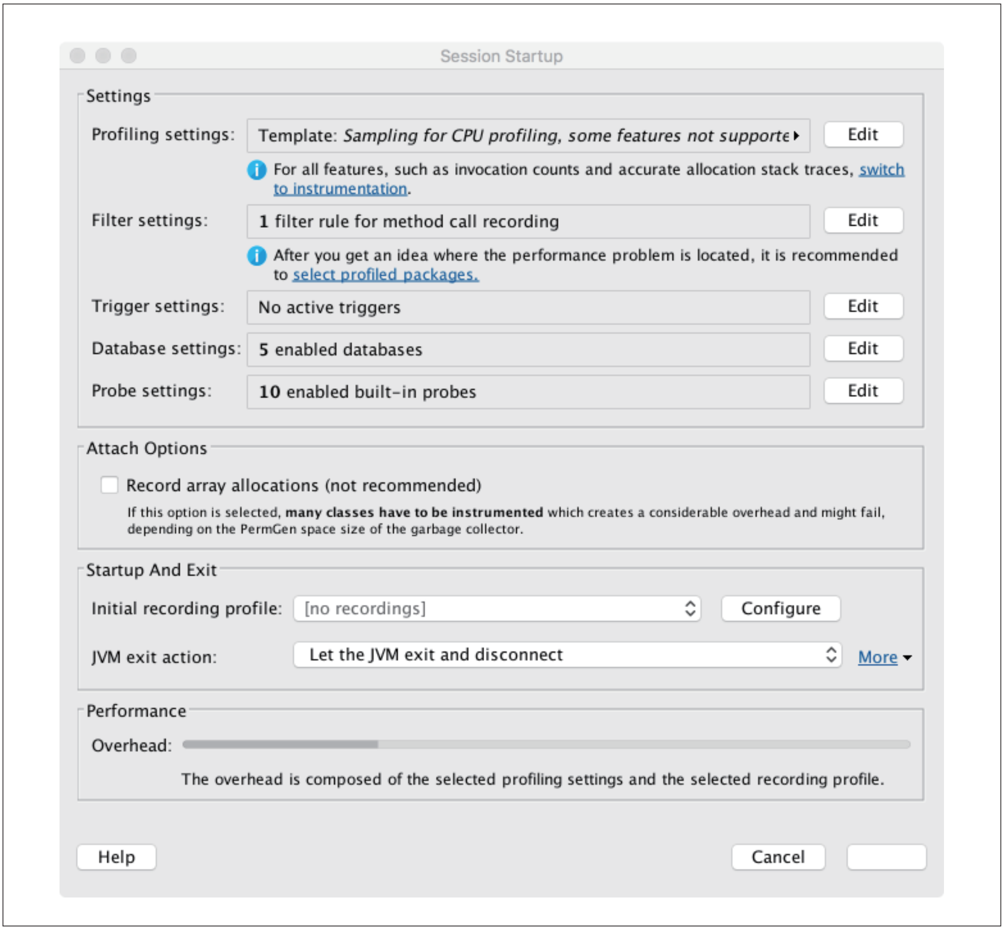


图 13-5：JProfiler 附加配置

在进行完初始扫描之后，JProfiler 就正式启动了。初始界面会显示一个类似于 VisualVM 的遥测视图，但这里可以滚动显示，而不是基于时间的变化调整视图，如图 13-6 所示。

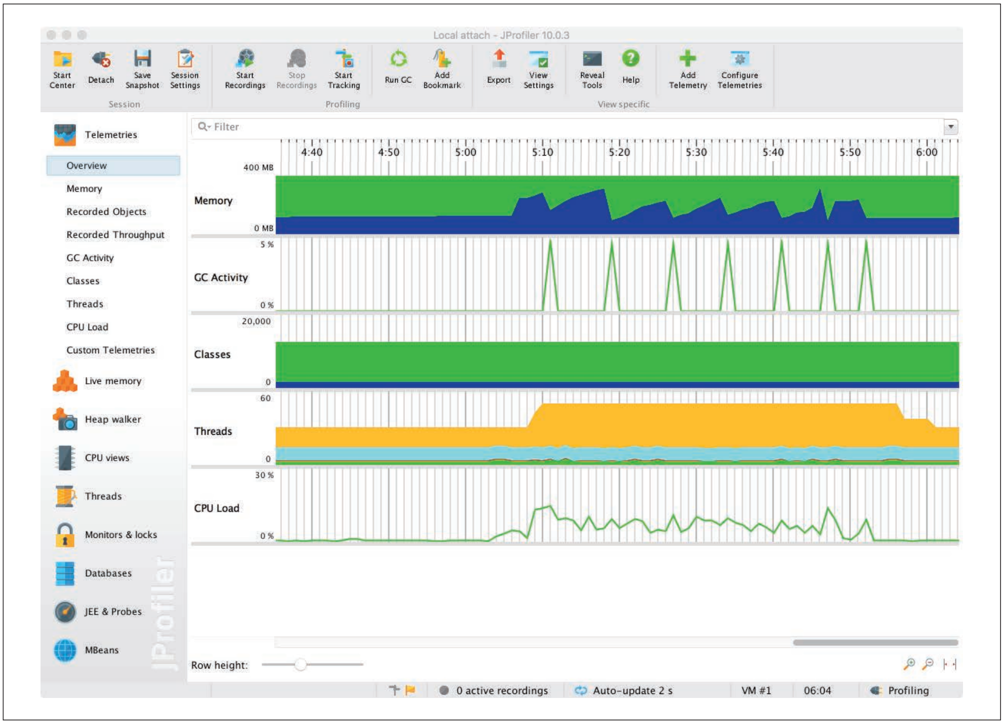


图 13-6：JProfiler 简单遥测视图

从这个界面可以访问所有的基本视图，但是如果不打开某些录制，能看到的不多。要查看方法计时，可以选择调用树（call tree）视图，然后点击按钮开始录制。几秒后，剖析的结果将开始显示出来，效果类似图 13-7。

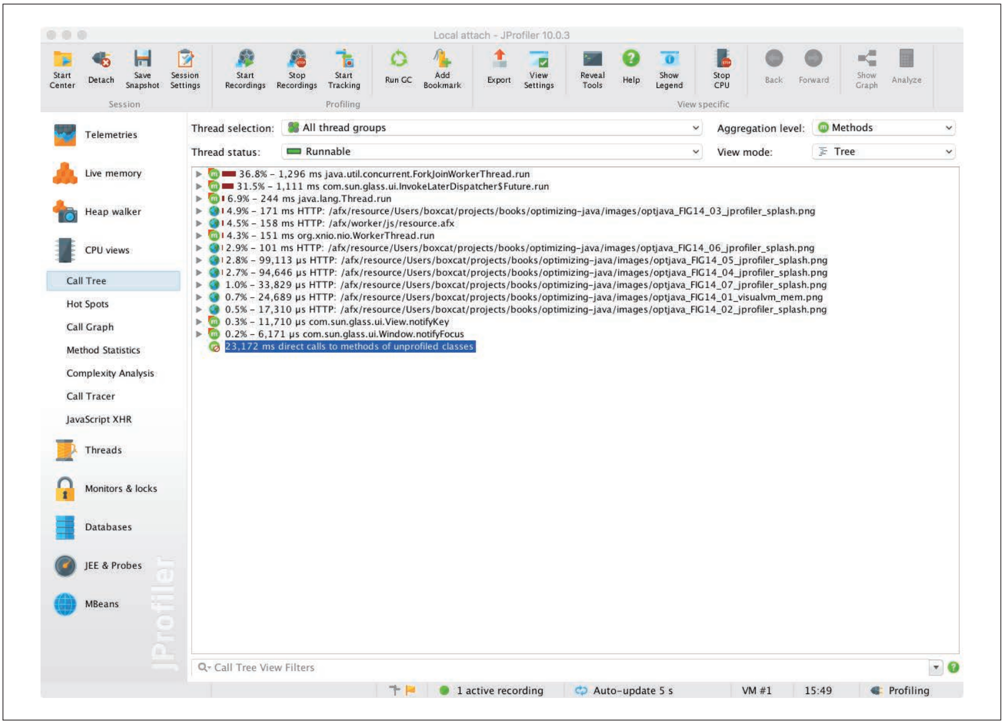


图 13-7：JProfiler CPU 时间

这个树状视图可以展开，显示每个方法调用的所有方法的内部时间。

要使用 JProfiler 代理，请将如下开关添加到运行配置中：

```
-agentpath:<path-to-agent-lib>
```

在默认配置下，这会引起被剖析的应用程序在启动时停顿，等待一个 GUI 来连接。这样做的目的是将对应用类的注入前置到启动时，之后应用程序就可以正常运行了。

对于生产应用程序，再附加一个 JProfiler GUI 就不太正常了。此时，需要添加的是 JProfiler 代理，并通过配置说明要记录哪些数据。这些结果仅保存到快照文件里，之后再加载到 JProfiler GUI 中。JProfiler 提供了一个向导，用于配置远程剖析，以及将适当的配置添加到远程的 JVM 中。

最后，细心的读者应该会注意到，截图中演示的是对一个 GUI 应用程序的剖析，它不是 CPU 密集型的，所以结果仅用于演示。CPU 的利用率远没有达到 100%，因此这并不是 JProfiler 或任何其他剖析工具的现实用例。

13.3.3　YourKit

YourKit 剖析器是另一款商用剖析器，由 YourKit 有限责任公司开发。YourKit 工具在某些方面和 JProfiler 类似，提供了一个 GUI 组件和一个代理，既可以动态附加，也可以通过配

置在应用程序启动时运行。

要部署其代理，使用如下语法（64 位 Linux）：

```
-agentpath:<profiler-dir>/bin/linux-x86-64/libyjpagent.so
```

从 GUI 的角度看，它的设置和初始遥测界面非常类似于 VisualVM 和 JProfiler 中的。

在图 13-8 中可以看到 CPU 快照视图，从中可以深挖出 CPU 时间的使用情况，其细节程度可能是 VisualVM 所达不到的。

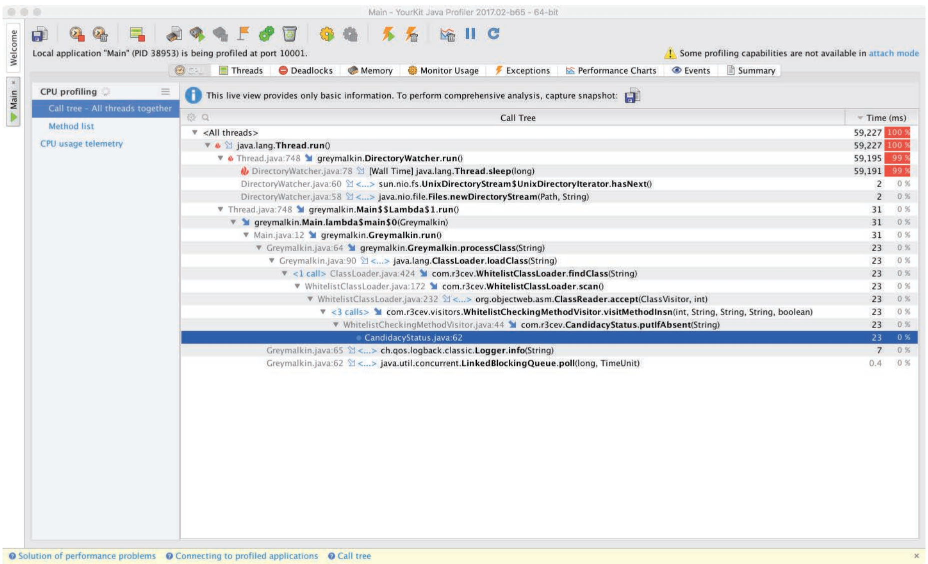


图 13-8：YourKit CPU 时间

在测试中，YourKit 的附加模式偶尔会出现些小问题，比如把 GUI 应用程序卡住了。不过总的来说，YourKit 的执行剖析功能大致上和 JProfiler 处于同等水平，有些工程师可能会出于个人偏好更喜欢其中的某一款。

如果可能的话，可以使用 YourKit 和 JProfiler 这两款工具进行剖析（不能同时进行，因为这会带来额外的开销），因为通过它们可能有不同的发现，这对诊断是有帮助的。

由于这两种工具都使用了前面讨论过的安全点采样方法，因此它们可能都存在由该方法引入的同样的局限性和偏差。

13.3.4　Java Flight Recorder和Java Mission Control

Java Flight Recorder（JFR）和 Java Mission Control（JMC）工具是 Oracle 收购 BEA System 而获得的剖析和监控技术。它们之前是用于 BEA 的 JRockit JVM 的工具产品的一部分。在 JRockit 退出市场的过程中，这些工具被移植到了 Oracle JDK 的商业版本中。

在 Java 8 之前，Flight Recorder 和 Mission Control 都是商用的专有工具。它们仅为 Oracle JVM 提供，不能与 OpenJDK 构建版本或任何其他 JVM 配合使用。

因为 JFR 仅为 Oracle JDK 提供，所以在启动带有 Flight Recorder 的 Oracle JVM 时，必须传入如下开关：

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

2017 年 9 月，Oracle 宣布对 Java 的发布时间表进行重大调整，将其从两年的发布周期改为 6 个月。这是由于前两个版本（Java 8 和 Java 9）严重延期而导致的。

除了决定改变发布周期外，Oracle 还宣布，在 Java 9 之后，其发布的主要 JDK 将变为 OpenJDK，而不是 Oracle JDK。作为这一变化的一部分，Flight Recorder 和 Mission Control 会成为可以免费使用的开源工具。

在编写本书时，JFR/JMC 成为免费和开源特性的详细路线图尚未公布。对于 Java 8 和 Java 9 的部署，如果要在产品中使用 JFR/JMC，那么是否需要付费，也没有公布。

> JMC 的初始安装包括一个 JMX 控制台和 JFR，不过在 Mission Control 内安装更多插件也非常容易。

JMC 是图形组件，从 `jmc` 二进制文件 $JAVA_HOME/bin 中启动。Mission Control 的启动界面如图 13-9 所示。

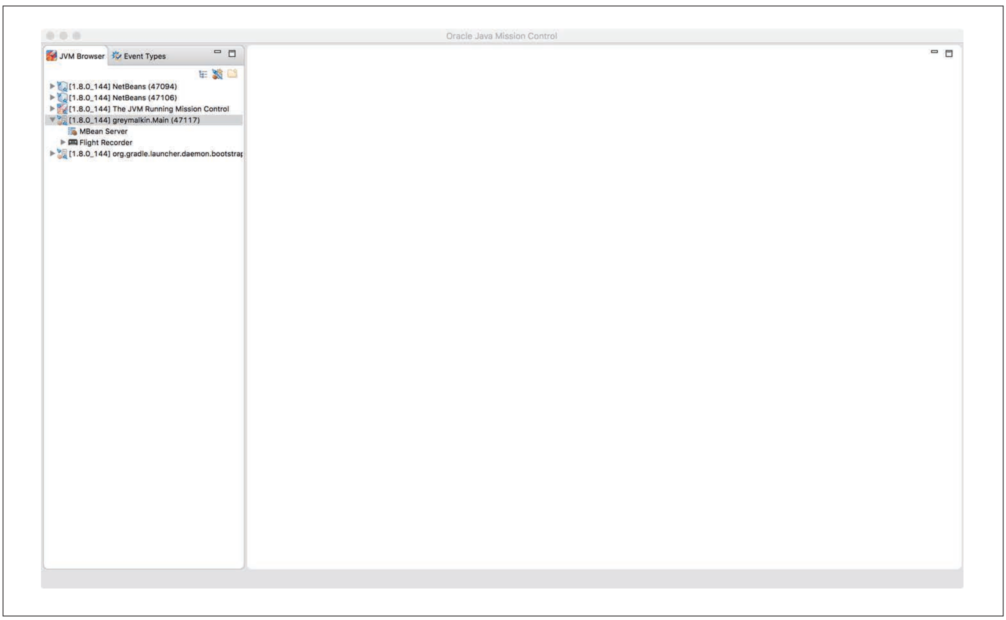


图 13-9：JMC 启动界面

要进行剖析，必须在目标应用程序上启动 Flight Recorder。你可以在启动时使用相关标志或在应用程序启动后通过动态附加来实现这一点。

连接后，输入录制会话和剖析事件的配置，如图 13-10 和图 13-11 所示。

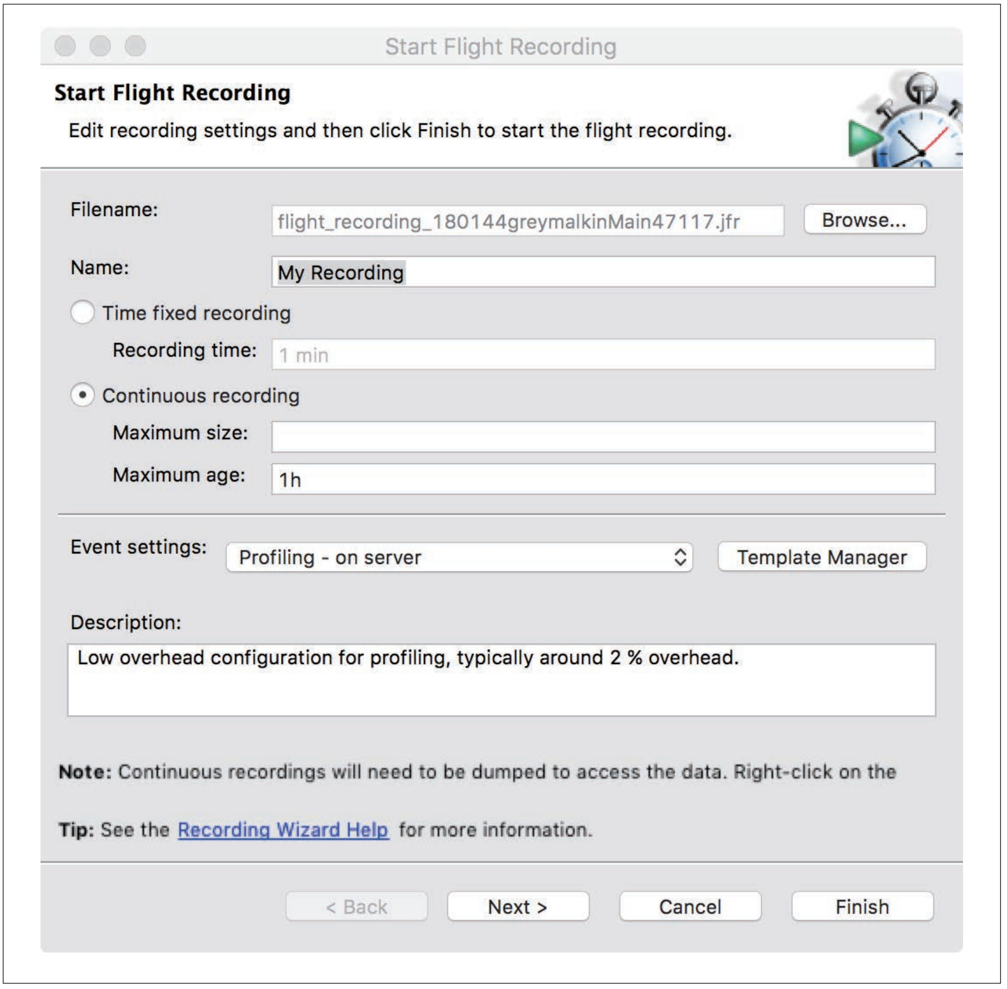


图 13-10：JMC 录制设置

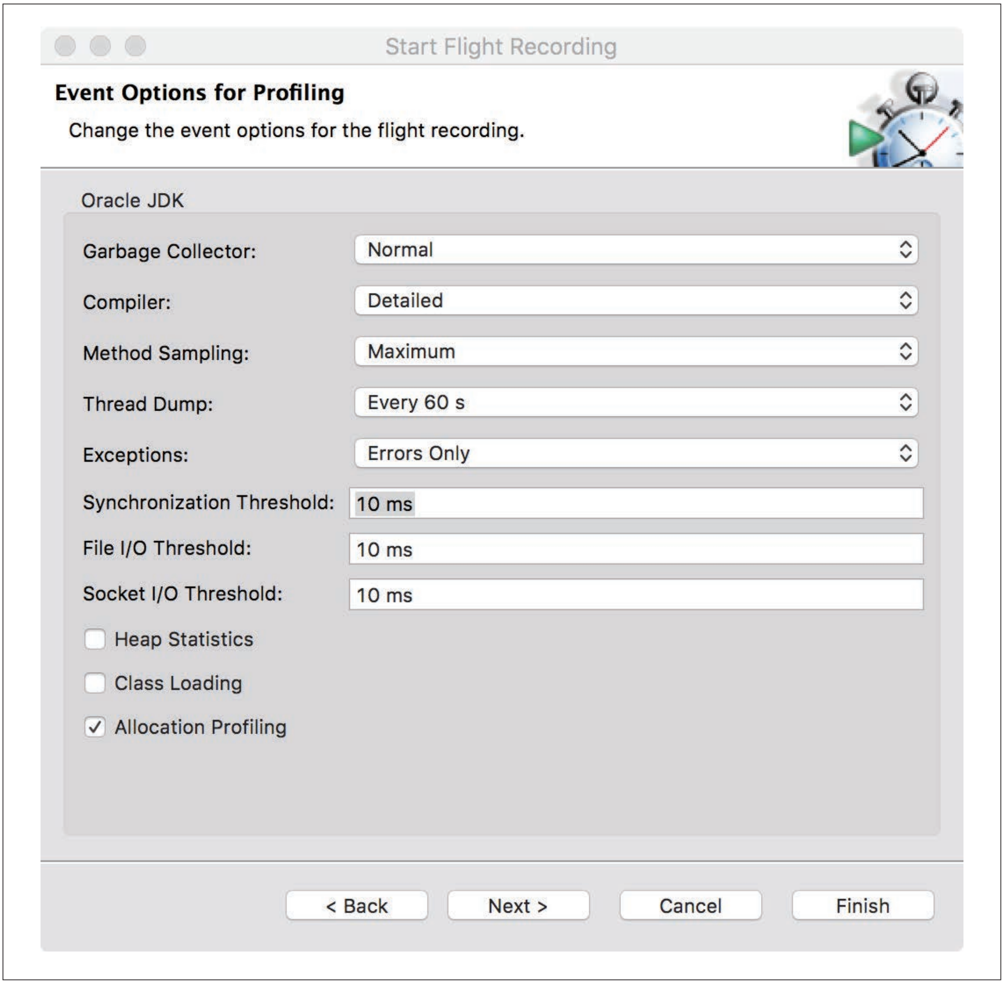


图 13-11：JMC 剖析事件选项

当录制启动时，通常会显示在一个时间窗口中，如图 13-12 所示。



图 13-12：JMC 时间窗口

为了支持将 JFR 从 JRockit 移植过来，HotSpot 虚拟机中加入了大量的性能计数器，它们与 Serviceability Agent 中收集的那些类似。

13.3.5 运维工具

就其本质而言，剖析器是供开发人员使用的工具，用于在底层诊断问题或了解应用程序的运行时行为。在工具谱的另一端是运维监控工具，它们帮助团队以可视化的方式显示系统的当前状态，并确定系统的运行正常与否。

这是一个复杂的问题，本书不会对其进行完整的讨论。相反，我们会简要介绍这个领域中的 3 个工具，其中两个是专有的，一个是开源的。

1. Red Hat Thermostat

Thermostat 是 Red Hat 针对基于 HotSpot 的 JVM 的开源的服务能力和监控的解决方案。它与 OpenJDK 本身使用相同的许可证，并提供了针对单机和集群的监控。它使用 MongoDB 来存储历史数据以及时间点（point-in-time）功能。

Thermostat 被设计为一个开放的、可扩展的平台，由代理和客户端（通常是一个简单的 GUI）组成。图 13-13 是一个简单的 Thermostat 视图。

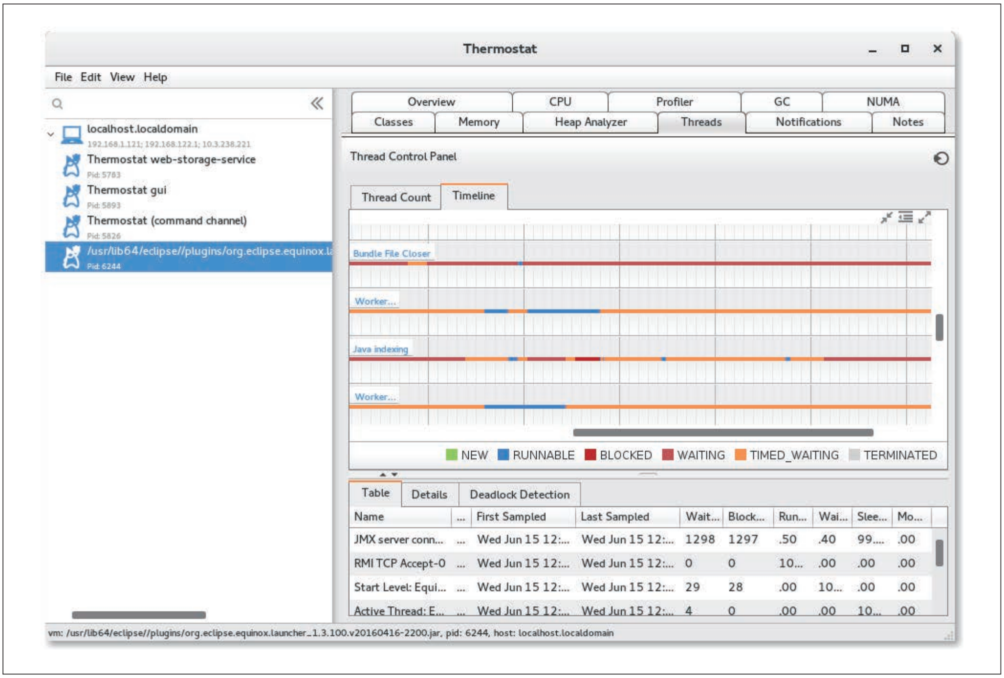


图 13-13：Red Hat Thermostat

Thermostat 的架构支持扩展，比如你可以：

- 收集和分析自定义的指标；
- 按需注入定制代码；
- 编写自定义插件和集成工具。

Thermostat 的大部分内置功能实际上就是以插件形式实现的。

2. New Relic

New Relic 工具是为基于云的应用设计的一款 SaaS 产品。它是通用的工具集，不仅仅支持 JVM。

在 JVM 的支持方面，安装时需要下载代理，向 JVM 传递一个开关，并重启服务器。按照这个步骤，New Relic 会生成与图 13-14 类似的视图。

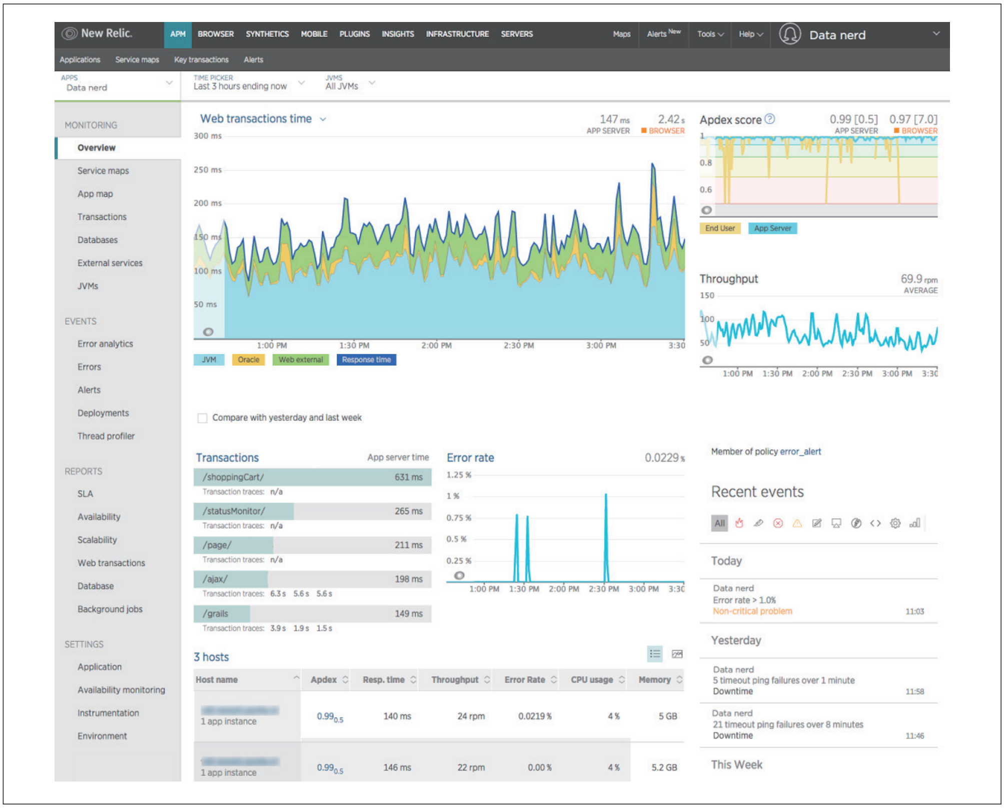


图 13-14：New Relic

New Relic 提供的通用监控和全栈支持使其成为一个很有吸引力的运维和 DevOps 工具。然而，作为一个通用工具，它并没有特别关注 JVM 技术，而是依赖于由 JVM 直接提供的不那么精细的数据。这意味着对于 JVM 的深层信息，它可能需要与更专用的工具结合。

New Relic 提供了一个 Java 代理 API，或者用户可以实现自定义注入来扩展其基础功能。

它还存在产生的数据量特别大这一问题，所以有时候除了输出明显增加这个趋势外，很难再发现任何其他东西。

3. jClarity Illuminate

jClarity Illuminate 在开发人员剖析工具和运维监控工具之间提供了一个桥梁。这不是一个传统的采样剖析器，相反，它以监控模式运行，通过进程之外的一个单独的守护线程来观察主 Java 应用程序。如果 Illuminate 监测到运行中的 JVM 的行为异常，比如违反了服务等级协议（SLA），就将启动对应用程序的深度监测。

Illuminate 的机器学习算法会分析从操作系统、垃圾收集日志和 JVM 收集到的数据，从而确定出现性能问题的根本原因。它会生成一份详细的报告，并将其发送给用户，同时附上一些可能解决问题的下一步方法。机器学习算法基于性能诊断模型（performance diagnostic model，PDM），该模型最初由 jClarity 的创始人之一 Kirk Pepperdine 创建。

在图 13-15 中，可以看到 Illuminate 在调查自动发现的问题时的**分流模式**（triage mode）。

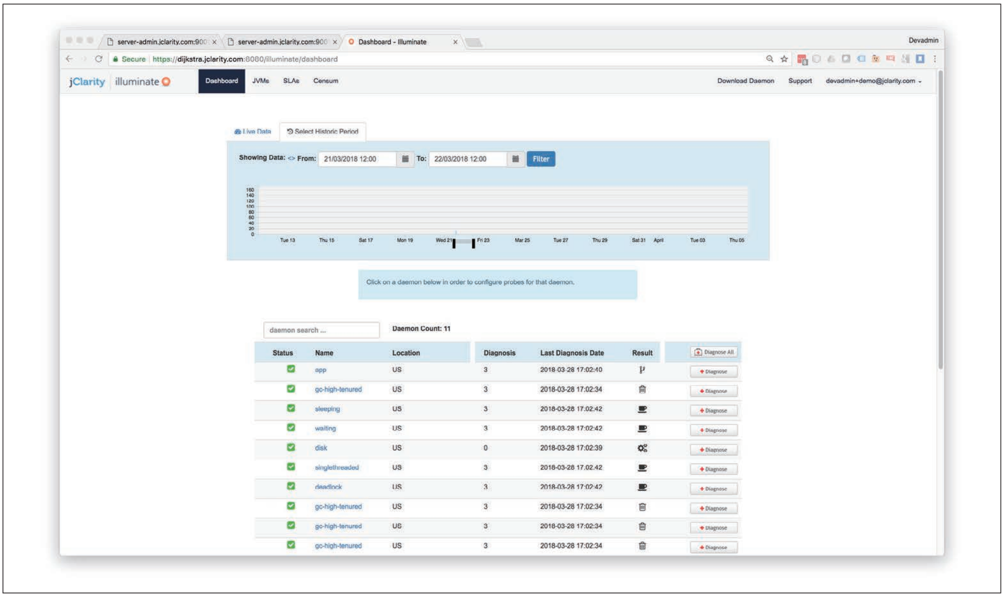


图 13-15：jClarity Illuminate

该工具基于机器学习技术，专注于深入地分析根本原因，不像其他监控工具那样有时还会遇到“数据墙”。它还大幅减少了需要收集、在网络上移动和存储的数据量，希望与其他性能监控工具相比，它能减少对应用程序的影响。

13.4　现代剖析器

本节将讨论 3 个现代的开源工具，与传统剖析器相比，它们可以提供更好的分析和更准确的性能数据。这些工具是：

- Honest Profiler
- perf
- Async Profiler

Honest Profiler 是一个相对较新的剖析工具。它是由 Richard Warburton 领导的开源项目，源于 Google 工程团队的 Jeremy Manson 编写并开源的原型代码。

Honest Profiler 的主要目标如下：

- 去除大多数其他剖析器存在的安全点偏差；
- 显著降低自身的运行开销。

为了实现这一点，它使用了 HotSpot 中的一个叫作 `AsyncGetCallTrace` 的私有 API。当然，这也就意味着 Honest Profiler 不能在 HotSpot 之外的其他 JVM 上运行，但可以在 Oracle、Red Hat 和 Azul Zulu JVM 等基于 HotSpot 构建的 JVM 上运行。

该实现使用 Unix 操作系统信号 SIGPROF 来中断正在运行的线程。然后可以通过私有的 `AsyncGetCallTrace()` 方法收集调用栈信息。

它只会单独中断线程，所以永远不会有任何形式的全局同步事件，进而也就避免了传统剖析器中常见的争用和开销。在异步回调中，调用轨迹（call trace）被写入一个无锁的环形缓冲区（ring buffer）。然后，会有一个专门的独立线程在不暂停应用程序的情况下将详细信息写入日志。

> Honest Profiler 并不是唯一采用这种方法的剖析器。Flight Recorder 也使用了 `AsyncGetCallTrace()` 调用。

历史上，Sun Microsystems 公司还提供了比较陈旧的 Solaris Studio 产品，它也使用了私有 API 调用。遗憾的是，该产品的名称令人相当困惑，因为它实际上也能在其他操作系统上运行，而不仅仅是 Solaris，所以它未能得到应用。

Honest Profiler 有个缺点，即在某些线程上面，它可能会显示“Unknown”。这是 JVM 内部函数的副作用，在这种情况下，剖析器无法映射回真正的 Java 栈轨迹。

要使用 Honest Profiler，必须安装该剖析器的代理：

```
-agentpath:<path-to-liblagent.so>=interval=<n>,logPath=<path-to-log.hpl>
```

Honest Profiler 包含一个用来使用剖析器的相当简单的 GUI。它基于 JavaFX，所以如果需要在基于 OpenJDK 的构建版本（比如 Azul Zulu 或 Red Hat IcedTea）上运行，就应该安装 OpenJFX。

图 13-16 是一个典型的 Honest Profiler GUI 界面。

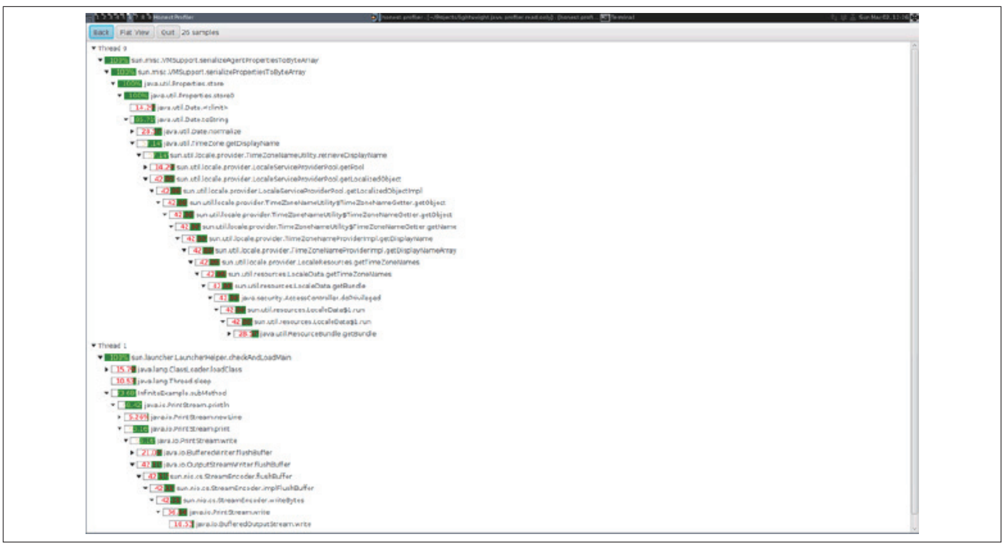


图 13-16：Honest Profiler

在实践中，像 Honest Profiler 这样的工具更多是作为数据收集工具以 Headless 模式运行。如果这样使用，可视化则由其他工具或自定义脚本来提供。

该项目确实提供了二进制文件，但使用范围相当有限，仅适用于最新的 Linux 构建版本。大多数严谨的 Honest Profiler 用户需要从零开始构建自己的二进制文件，不过这个话题超出了本书的讨论范围。

perf 工具是一个有用的轻量级剖析解决方案，适用于运行在 Linux 上的应用程序。它不是专用于 Java/JVM 应用程序的，其原理是读取硬件性能计数器，它包含在 Linux 内核中，位于 tools/perf 下。

性能计数器是物理寄存器，可以计算性能分析人员感兴趣的硬件事件的数目。这些事件包括执行的指令、高速缓存未命中和分支预测错误，它们构成了对应用程序进行剖析的基础。

由于 Java 运行时环境的动态特性，使得 Java 给 perf 带来了一些额外的挑战。为了在 Java 应用程序中使用 perf，我们需要一个桥梁来处理 Java 执行的动态部分的映射。

这个桥梁就是 perf-map-agent，它是一个代理，可以为来自未知内存区域（包括 JIT 编译的方法）的信息生成动态符号。由于 HotSpot 有动态创建的解释器和用于虚拟分派的跳转表，因此这些也必须有生成的条目来映射。

perf-map-agent 由用 C 语言编写的代理和小的 Java 引导程序组成，如果需要的话，那这个代理可以附加到一个正在运行的 Java 进程中。Java 8u60 中有一个新的标志，支持与 perf 进行更好的交互：

```
-XX:+PreserveFramePointer
```

当使用 perf 来剖析 Java 应用程序时，强烈建议在 8u60 或更高版本上运行，以访问该标志。

> 因为使用这个标志会禁用 JIT 编译器优化，所以它确实会稍微降低性能（在测试中最多降低 3%）。

火焰图（flame graph）用可视化的方式将 perf 生成的数据醒目地显示出来。对于执行时间到底花在了哪里，火焰图显示了非常详细的分类，示例如图 13-17 所示。

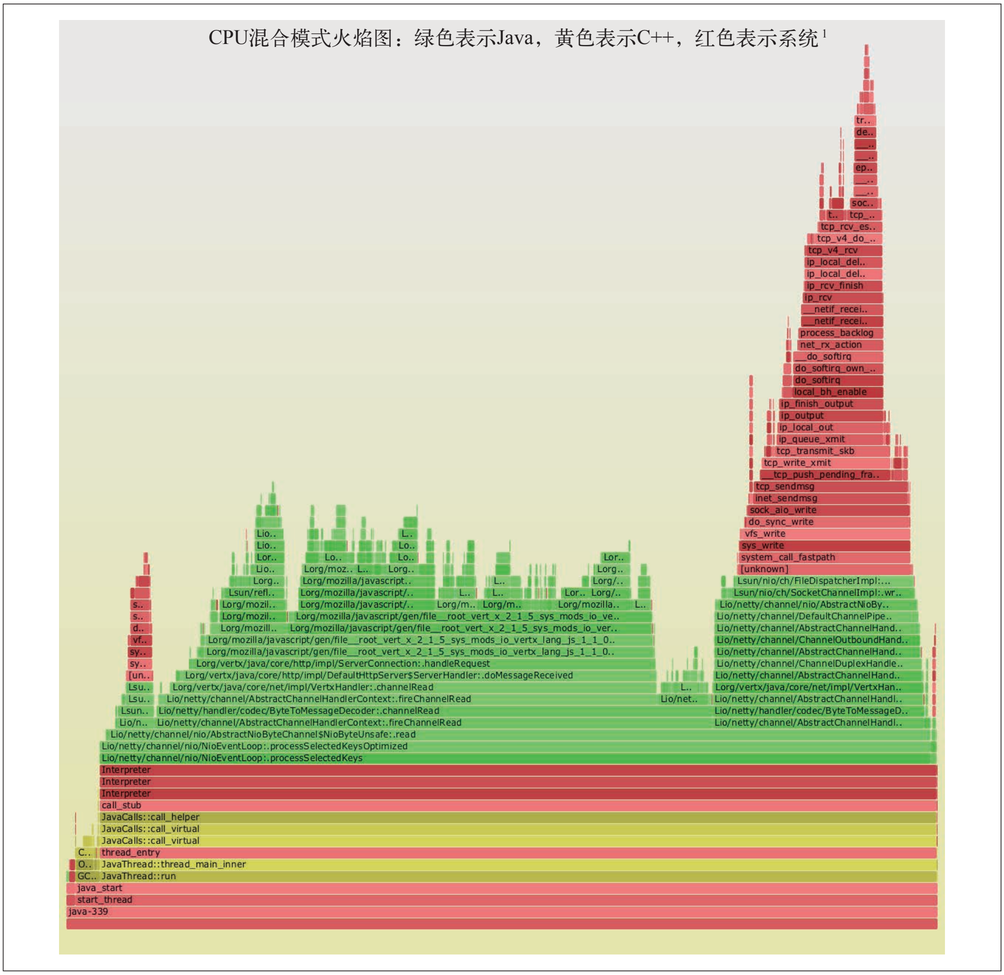


图 13-17：Java 火焰图

Netflix 技术博客上有一些文章详细介绍了他们团队是如何在自己的 JVM 上实现火焰图的。

最后，可以替代 Honest Profiler 的另一个选择是 Async Profiler。它使用了与 Honest Profiler 同样的内部 API，这是一个只运行在 HotSpot JVM 上的开源工具。它依赖于 perf，这意味着 Async Profiler 也只能运行在支持 perf 的操作系统上（主要是 Linux）。

13.5　分配剖析器

执行剖析是剖析的一个重要方面，但并不是唯一的方面！大多数应用程序也需要某种程度的内存剖析，考虑应用程序的分配行为就是一个标准的方法。分配剖析有几种可能的方法。

注 1：本书彩图可到图灵社区本书主页（ituring.cn/book/2085）“随书下载”处查看。——编者注

比如，可以使用 HeapVisitor 方法，像 jmap 等工具就依赖于它。在图 13-18 中可以看到 VisualVM 的内存剖析视图，它就使用了这种简单的方法。

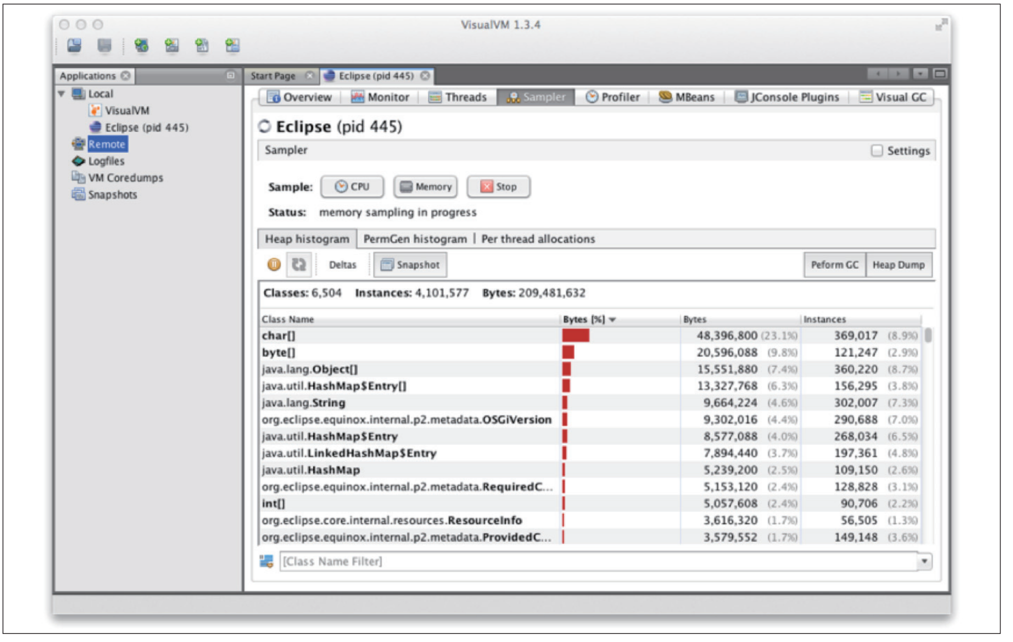


图 13-18：VisualVM 内存剖析器

图 13-19 显示了类似的视图，其中我们使用 YourKit 分配剖析器展示了同样的功能。

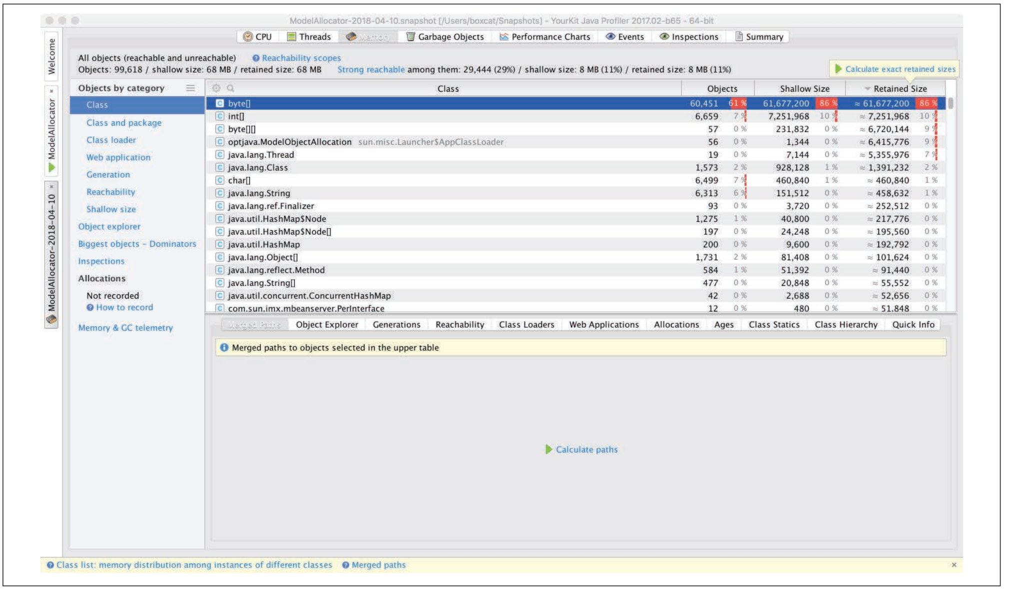


图 13-19：YourKit 内存剖析器

对于 JMC 工具来说，垃圾收集统计信息中包含了传统的 Serviceability Agent 中没有的一些值，不过所显示的绝大多数计数器在两者中存在。其优势在于 JFR 收集这些值的成本，所以在 JMC 中显示要比在 SA 中显示代价更小。在如何显示细节方面，JMC 显示也为性能工程师提供了更大的灵活性。

分配剖析的另一种方法是基于代理进行剖析，它可以通过几种方式来完成，其中最简单的一种是对字节码进行注入。正如在 9.1.1 节所看到的，指示 JVM 分配内存的字节码有以下 3 个。

`NEW`

　为一个指定类型的新对象分配空间。

`NEWARRAY`

　为一个基本类型的数组分配空间。

`ANEWARRAY`

　为指定类型对象的一个数组分配空间。

只需要对这 3 个字节码进行注入，因为只有它们才会引起分配行为。

一个简单的注入方法包含这样两个部分：首先定位到每个使用了任何一个分配操作码的实例，其次插入一个对某个静态方法的调用，该方法会在分配操作码执行之前将该分配写入日志。

接下来看看这样一个分配剖析器的框架。我们需要通过一个 `premain()` 钩子使用 Instrumentation API 来设置代理：

```
public class AllocAgent {

    public static void premain(String args, Instrumentation instrumentation) {
        AllocRewriter transformer = new AllocRewriter();
        instrumentation.addTransformer(transformer);
    }

}
```

通常要使用一个库来实现这种字节码注入，而不是试图用手写的代码来实现变换。一个广泛使用的常见字节码操作库是 ASM，我们将用它来演示分配剖析。

为了添加所需要的分配注入代码，需要一个类重写器。它提供了 Instrumentation API 和 ASM 之间的桥梁，看起来像这样：

```
public class AllocRewriter implements ClassFileTransformer {

    @Override
    public byte[] transform(ClassLoader loader, String className,
      Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
      byte[] originalClassContents) throws IllegalClassFormatException {
        final ClassReader reader = new ClassReader(originalClassContents);
        final ClassWriter writer = new ClassWriter(reader,
            ClassWriter.COMPUTE_FRAMES | ClassWriter.COMPUTE_MAXS);
        final ClassVisitor coster = new ClassVisitor(Opcodes.ASM5, writer) {
            @Override
```

```
            public MethodVisitor visitMethod(final int access, final String name,
                final String desc, final String signature,
                final String[] exceptions) {
                final MethodVisitor baseMethodVisitor =
                    super.visitMethod(access, name, desc, signature, exceptions);
                return new AllocationRecordingMethodVisitor(baseMethodVisitor,
                    access, name, desc);
            }
        };
        reader.accept(coster, ClassReader.EXPAND_FRAMES);
        return writer.toByteArray();
    }

}
```

它使用一个方法访问者来检查字节码，并插入注入调用，以支持跟踪分配：

```
public final class AllocationRecordingMethodVisitor extends GeneratorAdapter {
    private final String runtimeAccounterTypeName =
        "optjava/bc/RuntimeCostAccounter";

    public AllocationRecordingMethodVisitor(MethodVisitor methodVisitor,
        int access, String name, String desc) {
        super(Opcodes.ASM5, methodVisitor, access, name, desc);
    }

    /**
     * 当访问到一个带有单个int操作数的操作码时，该方法被调用
     * 就目的而言，这是一个NEWARRAY操作码
     *
     * @param opcode
     * @param operand
     */
    @Override
    public void visitIntInsn(final int opcode, final int operand) {
        if (opcode != Opcodes.NEWARRAY) {
            super.visitIntInsn(opcode, operand);
            return;
        }

        // 操作码是NEWARRAY - recordArrayAllocation:(Ljava/lang/String;I)V
        // 操作数的值应该是这几种类型之一：Opcodes.T_BOOLEAN、
        // Opcodes.T_CHAR、Opcodes.T_FLOAT、Opcodes.T_DOUBLE、Opcodes.T_BYTE、
        // Opcodes.T_SHORT、Opcodes.T_INT或Opcodes.T_LONG
        final int typeSize;
        switch (operand) {
            case Opcodes.T_BOOLEAN:
            case Opcodes.T_BYTE:
                typeSize = 1;
                break;
            case Opcodes.T_SHORT:
            case Opcodes.T_CHAR:
                typeSize = 2;
                break;
            case Opcodes.T_INT:
```

```
            case Opcodes.T_FLOAT:
                typeSize = 4;
                break;
            case Opcodes.T_LONG:
            case Opcodes.T_DOUBLE:
                typeSize = 8;
                break;
            default:
                throw new IllegalStateException("Illegal op: to NEWARRAY seen: "
                    + operand);
        }
        super.visitInsn(Opcodes.DUP);
        super.visitLdcInsn(typeSize);
        super.visitMethodInsn(Opcodes.INVOKESTATIC, runtimeAccounterTypeName,
            "recordArrayAllocation", "(II)V", true);
        super.visitIntInsn(opcode, operand);
    }

    /**
     * 当访问到一个操作数为类型（这里表示为一个String）的操作码时，该方法就被调用
     *
     * 就目的而言，这个操作码或者是NEW，或者是ANEWARRAY
     *
     * @param opcode
     * @param type
     */
    @Override
    public void visitTypeInsn(final int opcode, final String type) {
        // 操作码是NEW - recordAllocation:(Ljava/lang/String;)V
        // 或ANEWARRAY - recordArrayAllocation:(Ljava/lang/String;I)V
        switch (opcode) {
            case Opcodes.NEW:
                super.visitLdcInsn(type);
                super.visitMethodInsn(Opcodes.INVOKESTATIC,
                    runtimeAccounterTypeName, "recordAllocation",
                    "(Ljava/lang/String;)V", true);
                break;
            case Opcodes.ANEWARRAY:
                super.visitInsn(Opcodes.DUP);
                super.visitLdcInsn(8);
                super.visitMethodInsn(Opcodes.INVOKESTATIC,
                runtimeAccounterTypeName, "recordArrayAllocation",
                    "(II)V", true);
                break;
        }

        super.visitTypeInsn(opcode, type);
    }
}
```

这也需要一个小的运行时组件：

```
public class RuntimeCostAccounter {
    private static final ThreadLocal<Long> allocationCost =
        new ThreadLocal<Long>() {
```

```
            @Override
            protected Long initialValue() {
                return 0L;
            }
        };

        public static void recordAllocation(final String typeName) {
            // 显然要更复杂一些
            // 比如缓存我们遇到的类型的近似大小
            checkAllocationCost(1);
        }

        public static void recordArrayAllocation(final int length,
            final int multiplier) {
            checkAllocationCost(length * multiplier);
        }

        private static void checkAllocationCost(final long additional) {
            final long newValue = additional + allocationCost.get();
            allocationCost.set(newValue);
            // 采取某个动作？比如，如果超过某个阈值，则失败
        }

        // 这可以暴露出去，比如通过一个JMX计数器
        public static long getAllocationCost() {
            return allocationCost.get();
        }

        public static void resetCounters() {
            allocationCost.set(0L);
        }
    }
```

这两组代码的目的是提供一个简单的分配注入功能。它们使用了 ASM 字节码操作库（很遗憾，完整描述这个库超出了本书的范围）。方法访问者在字节码 `NEW`、`NEWARRAY` 和 `ANEWARRAY` 的每一个实例之前都添加了对记录方法的调用。

进行这种变换之后，每当有任何新的对象或数组创建时，该记录方法就会被调用。这必须通过一个很小的类在运行时提供支持——`RuntimeCostAccounter`（它必须在类路径上）。这个类会记录在每个线程中被注入的代码分配了多少内存。

虽然上面提供的这种字节码级的技术还相当粗糙，但是这给有兴趣开发自己的工具来测量其线程分配了多少内存的读者提供了一个起点。比如，这可以用于单元测试或回归测试，以确保对代码的修改不会引入大量的额外分配。

但是如果要完全应用于生产之中，这种方法可能并不合适。如果每次有内存分配就进行额外的方法调用，就会导致大量的额外调用。JIT 编译将对此有所帮助，因为注入的调用将被内联，但总体效果仍然可能会对性能产生非常大的影响。

还有一种分配剖析方法是 TLAB 耗尽。比如 Async Profiler 的特色就是基于 TLAB 的采样功能。它使用了 HotSpot 特有的回调功能来接收如下通知：

- 当一个对象在新创建的 TLAB 中分配时；
- 当一个对象在 TLAB 之外分配时（慢路径）。

因此，并不是每个对象的分配都会被计算在内。相反，总的来说，每 *n* KB 的分配都会被记录下来，其中 *n* 是 TLAB 的平均大小（回想一下，TLAB 的大小可能会随着时间的流逝而改变）。

这样设计的目的是使堆采样的开销足够低，以适合在生产中应用。但是，因为收集到的数据是采样过的，所以可能并不完整。其目的是在实践中它将反映出排在最上面的分配来源，所以对性能工程师而言，至少大部分时间是有用的。

要使用 TLAB 采样功能，我们需要使用 7u40 或更高版本的 HotSpot JVM，因为 TLAB 回调首先会出现在这个版本中。

13.6 堆转储分析

堆转储分析（heap dump analysis）是与分配剖析相关的一种技术。该技术会使用工具来检查整个堆的快照，并确定最重要的事实，比如活跃集合、对象的类型和数量，以及对象图的形状和结构。

加载了堆转储后，性能工程师就可以遍历和分析堆转储创建之时的堆快照。他们将能看到活跃的对象和任何已经死亡但尚未被收集的对象。

堆转储的主要缺点是其庞大的体积。一个堆转储经常可以达到被转储内存大小的 300%~400%。对于有几个 GB 的堆而言，这是相当大的。堆不仅必须被写入磁盘，而且对于真正的生产用例，它还需要通过网络检索。一旦被检索，它就必须被加载到有足够资源（特别是内存）的工作站上，以便在不向工作流程中引入过多延迟的情况下处理该转储。对于较大型的堆转储，在一台不能一次性将其加载进来的机器上处理非常痛苦，因为转储文件的部分内容需要在磁盘和内存之间进行页面交换。

要生成堆转储文件，当遍历堆并写出转储时，也需要一个 STW 事件。

YourKit 支持以 `hprof` 和专有格式抓取内存快照。图 13-20 是堆转储分析器的视图。

在商业工具中，对于堆转储的过滤器和其他视图，YourKit 提供了一个很好的选择，其中包括能够通过类加载器（class loader）和 Web 应用程序对堆进行分解，从而可以更快地诊断堆问题。

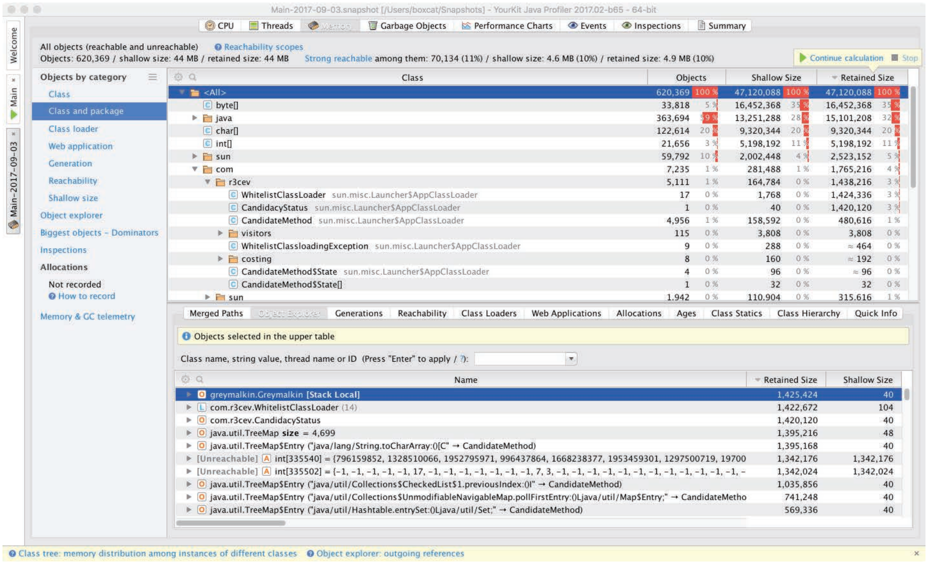


图 13-20：YourKit 内存转储分析器

JMC/JFR 的分配视图也是值得考虑的工具。它能够显示 TLAB 分配视图，Async Profiler 也有这个视图。图 13-21 是 JMC 分配视图的一个示例。

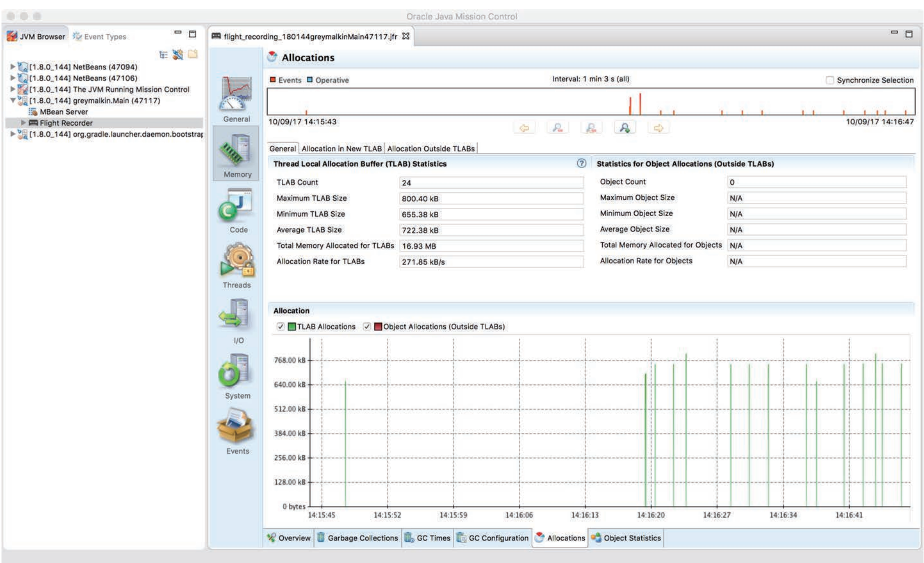


图 13-21：JMC 分配剖析视图

对于大部分需要剖析的应用程序来说，分配和剖析是应该考虑的，我们希望性能工程师不要因为过度关注执行剖析而忽视了内存剖析。

hprof

从 JDK 5 起，`hprof` 剖析器就随 JDK 一起发布了。其主要目的是作为 JVMTI 技术的参考实现，而不是一个生产级的剖析器。尽管如此，它经常在文档中被提及，使得一些开发人员认为 `hprof` 是一个适合实际使用的工具。

从 Java 9（JEP 240）开始，`hprof` 将从 JDK 中移除，相关 JEP 对此有如下说明。

> `hprof` 代理是作为 JVM 工具接口（JVM TI）的演示代码编写的，无意成为生产工具。

此外，`hprof` 的代码和文档中包含了一些一般形式的说明。

> 这是 JVM TI 接口和使用 BCI 的演示代码；它不是 JDK 的正式产品或正式部分。

因此，除了作为堆快照的遗留格式外，你不应该依赖 `hprof`。在 Java 9 以及可预见的未来，创建堆转储的能力需要继续保持。

13.7 小结

剖析是一个经常被开发人员误解的主题。虽然执行剖析和内存剖析都是必要的技术，但是性能工程师了解他们正在做什么以及为什么这样做也是非常重要的。简单盲目地使用工具会产生完全不准确或不相关的结果，并浪费大量的分析时间。

对现代应用程序进行剖析需要使用工具，这方面有大量的选项，包括商业的和开源的。

第 14 章

高性能日志和消息系统

在与 C++ 等语言比较时，使用 Java 和 JVM 有时被认为是一种权衡。Java 通过减少开发人员在日常开发中必须处理的低级问题，提高了生产力。人们认为的权衡就是以牺牲底层控制和原始性能为代价，通过更高级别的语言抽象来提高开发人员的生产效率。

C++ 采取的方法是，不管有什么新的语言特性，性能都不能有丝毫的影响。C++ 的理念使我们可以做到非常精细的控制，但代价是每个开发人员都必须手动管理资源或遵从恰当的习惯用法。

Java 平台采取的方法是开发人员不需要关心底层细节。不要低估自动内存管理的好处，它对生产力的提升是巨大的，作者已经用多年的时间见证过，一个 C++ 程序员稍有不慎就会造成的错误，进而带来的损失让我至今都心有余悸。

然而，在 JVM 上的垃圾收集以及其他更高级别的托管抽象，都会在性能方面造成一定程度的不确定性。当然，在对延迟很敏感的应用程序中，这种不确定性应该尽量减少。



这是否意味着 Java 和 JVM 不是一个适合于高性能系统的平台呢？

本章旨在探索开发人员在考虑高性能、低延迟应用程序时需要解决的一些常见问题，同时还将研究低延迟系统的设计方法和要求。对于低延迟、高性能的系统来说，日志和消息是两个核心的考虑因素。

任何一位 Java 开发人员都应该关注日志，因为一个可维护的 Java 系统通常包含大量的日志消息。特别是对于重视延迟的开发人员来说，日志更是尤为重要。幸运的是，人们在日志的开发和研究方面已经积累了大量经验，本章将加以探讨。

消息系统提供了近年来最成功的架构模式之一。它们通常处于低延迟系统的前沿，一般以每秒处理的消息数量来衡量。一个系统能够处理的消息数量往往是获得竞争优势的关键。如果能够把吞吐量（比如每秒的交易数量）和有形价值关联起来，那么在这个领域就可以获得广泛的研究和大量的资助。本章后面会讨论一些现代的消息方法。

14.1　日志

对于许多开发人员来说，他们并没有把日志库看作项目的重要组成部分，因此选择它们的时候会尽可能简单，不会考虑太多。这与向项目中引入的许多其他库形成了鲜明对比，人们会花时间研究其他库的功能，甚至可能要进行一些性能基准测试。

围绕生产级日志系统的选择，有以下几种反模式。

十年不变的日志记录器

　　曾经有人成功地配置过日志记录器，直接借用其配置要比重新创建容易得多。

项目范围的日志记录器

　　曾经有人对日志记录器进行了包装，以避免在项目的每个部分重新配置。

公司范围的日志记录器

　　曾经有人创建了一个日志记录器，供全公司使用。

当然不会**有人**刻意制造未来会出现的问题。对于日志架构的选择，通常有一个合理的理由，比如要和公司的其他功能集成，这就形成了公司范围的日志记录器。未来的问题往往与日志系统的维护有关，因为人们一般不会认为它对业务非常关键。这种忽视会导致可能波及整个组织的技术债务。尽管日志记录器不是那么让人兴奋，而且其选择往往也会按照前面提到的某种反模式进行，但是它们对所有应用程序都是非常重要的。

在很多高性能环境中，精确处理并报告与执行速度同样重要。速度很快，但是不正确是没有意义的，而且对精确报告处理的情况往往还有审计要求。日志有助于识别生产问题，所以应该记录足够多的信息，以便团队能够在事后调查问题。重要的是，不能简单地把日志记录器当作一项成本，而应该像系统中的任何其他组件一样，需要仔细控制并在深思熟虑之后引入项目。

对日志记录器做基准测试

本节将探讨一组微基准测试，旨在公平地比较 3 种最流行的日志记录器（Logback、Log4j 和 `java.util.logging`）在不同日志模式下的性能。

这些统计数据是基于 Stephen Connolly 的一个开源项目，可以在 GitHub 上找到。该项目设计得很好，是以可运行的基准测试套件的形式提供的，其中多个日志记录器可以在不同的配置下运行。

这些基准测试研究了每种日志记录器和不同日志格式的组合，从而使我们了解了日志框架的整体性能，以及模式是否会对性能产生任何影响。

这里有必要明确解释一下使用微基准测试方法的原因。在讨论这些具体技术的细节时，我们面临着一个与库的作者类似的问题：我们想了解不同日志记录器在不同配置下的性能，但知道要找到一组好的语料库，使其非常符合我们需要的日志配置，并提供可以进行有意义的比较的结果将是非常棘手的。

当代码在许多不同的应用程序中运行时，微基准测试可以估计代码将如何执行。这就是用于微基准测试的“通用的、不存在语料库意义的”用例。

这些数字提供了一个整体的概况，但在实际的应用程序中，使用的前后都必须对其进行剖析，以便真正理解修改带来的影响。

接下来看一些结果，以及它们是如何实现的。

1. 没有日志

没有日志是基准测试进行的一项测试，用于测试在没有日志操作的情况下的成本，它所生成的日志消息在日志记录器活跃的阈值之下。可以将其看作实验的控制组。

2. Logback 格式

```
14:18:17.635 [Name Of Thread] INFO  c.e.NameOfLogger - Log message
```

这个基准测试使用的是 Logback 的 1.2.1 版本。

3. `java.util.logging` 格式

```
Feb 08, 2017 2:09:19 PM com.example.NameOfLogger nameOfMethod
INFO: Log message
```

4. Log4j 格式

```
2017-02-08 14:16:29,651 [Name Of Thread] INFO com.example.NameOfLogger - message
```

这个基准测试使用的是 Log4j 的 2.7 版本。

5. 测量

为了比较，在 iMac 和 AWS（Amazon Web Services）EC t2.2xlarge 实例上已经运行了基准测试（参见表 14-1 和表 14-2）。在 macOS 上进行剖析时，可能会因为各种节能技术而导致问题；而 AWS 的缺点是其他容器可能会影响基准测试的结果。没有哪个环境是完美的：干扰总会存在，正如第 5 章所讨论的，微基准测试充满各种风险。希望通过比较这一基准测试的两个数据集可以帮助我们揭示一些有用的模式，为真正的应用程序的剖析提供指导。每当需要处理实验数据时，请记住 Feynman 的“不能糊弄自己”的原则。

表14-1：在iMac上执行的基准测试（ns/op）

	没有日志	Logback格式	`java.util.logging`格式	Log4j格式
Java util 日志记录器	158.051 (±0.762)	42 404.202 (±541.229)	86 054.783 (±541.229)	74 794.026 (±2244.146)
Log4j	138.495 (±94.490)	8056.299 (±447.815)	32 755.168 (±27.054)	5323.127 (±47.160)
Logback	214.032 (±2.260)	5507.546 (±258.971)	27 420.108 (±37.054)	3501.858 (±47.873)

表14-2：在AWS EC t2.2xlarge上执行的基准测试（ns/op）

	没有日志	Logback格式	`java.util.logging`格式	Log4j格式
Java util 日志记录器	1376.597 (±106.613)	54 658.098 (±516.184)	144 661.388 (±10 333.854)	109 895.219 (±5457.031)
Log4j	1699.774 (±111.222)	5835.090 (±27.592)	34 605.770 (±38.816)	5809.098 (±27.792)
Logback	2440.952 (±159.290)	4786.511 (±29.526)	30 550.569 (±39.951)	5485.938 (±38.674)

Java util 日志记录器在执行日志操作时，每个操作的时间是 42 404 ~86 054 纳秒。该日志记录器在使用 `java.util.logging` 格式时性能最差，与使用同样格式的 Log4j 相比，要差 2.5 倍以上。

总之，在 iMac 上运行该基准测试时，Logback 的性能最好，此时使用的是 Log4j 风格的日志格式化器。

通过观察在 AWS 上运行的基准测试，我们可以看到结果所表现出的整体模式与 iMac 上的类似。Logback 比 Log4j 稍快。从这些结果中可以得到一些关键的启示。



正确衡量对应用程序的影响，需要在生产硬件上进行配置修改的前后都对应用程序进行剖析。这里运行的基准测试应该在你实际中的生产机器上重复运行，不能照搬结论。

不过，在 AWS 上的整体执行速度明显更快，这可能是由于 iMac 上的节能功能或其他未被捕获的因素所致。

6. 日志记录器结果

从基准测试可以看出，根据所用的日志格式、日志框架和配置的不同，我们会得到一系列结果。从执行时间上看，util 日志一般来说总体表现最差，Log4j 格式似乎给出的结果最为一致，Logback 执行日志语句的时间最好。

在真实的系统中，在生产包上测试执行时间性能是值得的，特别是当数字如此接近时。真实系统难以捉摸，只有最清晰的信号才能作为任何底层信息的证据，通常情况下几十个百分点是不够的。

正如第 5 章所讨论的，微基准测试的危险在于，从小处研究问题可能会掩盖其对应用程序的整体影响，并且根据这些微基准测试所做出的决定可能还会给应用程序带来意想不到的影响。

就像消耗在记录日志而不是处理关键业务的并发任务上的 CPU 时间一样，日志框架所产生的垃圾的数量也是一个考量因素。日志库的设计和其工作机制，与微基准测试中的直线执行的结果同样重要。

14.2　设计一个影响较低的日志记录器

日志是任何应用程序的关键组件，但在低延迟的应用程序中，日志记录器不能成为业务逻辑性能的瓶颈，这是至关重要的。本章前面探讨过这样的看法，即开发人员往往不会刻意

地去选择合适的日志记录器，也不会做基准测试。在很多情况下，只有当日志记录成为应用程序中的一个巨大或主要的成本时，它才会作为一个问题浮出水面。

> 到目前为止，我遇到的客户中很少有人没有受到日志带来的某种负面影响。有个极端的案例，即客户有 4.5 秒的时间预算，日志记录就用了 4.2 秒。
>
> ——Kirk Pepperdine

Log4j 2.6 版本希望通过引入一个稳态的无垃圾日志记录器来解决 Kirk 提出的问题。

文档中重点列出了一个简单的测试，包括在 Java Flight Recorder 中运行应用程序，以便在 12 秒内尽可能频繁地对打印字符串日志的操作进行采样。日志记录器配置通过 `RandomAccessFile` 进行追加操作的异步方式，使用的模式为 `%d %p %c{1.} [%t] %m %ex%n`。

图 14-1 演示的是一个非稳态的垃圾收集器和一个用于比较的采样配置。这里无意提供一个精确的基准测试，而是希望对日志行为的剖析有个整体的了解。从剖析器可以很明显地看出垃圾收集周期：总计收集次数是 141 次，平均暂停时间约为 7 毫秒，最大暂停时间为 52 毫秒。

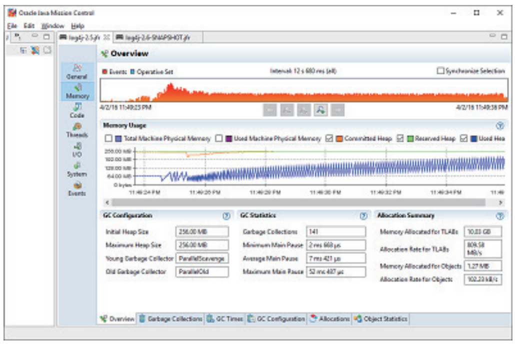


图 14-1：对 Log4j 2.5 的运行进行采样

将图 14-1 与图 14-2 进行比较，可以看出 Log4j 2.6 的不同之处：在相同的时间段内，没有出现垃圾收集周期。

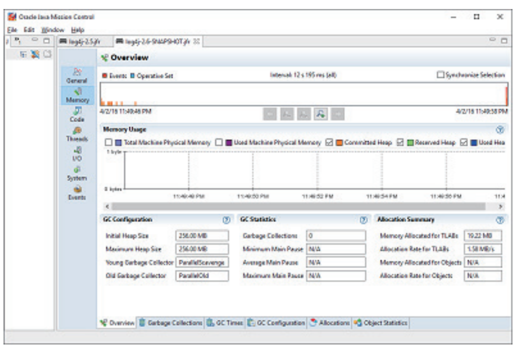


图 14-2：对 Log4j 2.6 的运行进行采样

当以图 14-2 所示的方式配置运行时，Log4j 2.6 具有明显的优势。然而，受零分配日志记录器实现的方式所限，它也有一些不足，毕竟天下没有免费的午餐。

我们看到 Log4j 的性能提升是通过复用对象，而不是为每条日志消息创建临时对象来实现的。这是对象池（object pool）模式的经典用法，各种后果也随之而来。Log4j 2.6 通过使用 `ThreadLocal` 字段和复用用于将字符串转换为字节的缓冲区来实现对象复用。

> 这也是我们仅仅通过观察微基准测试而无法推断出的结果之一。和以往一样，必须考虑设计和大局。

`ThreadLocal` 对象在 Web 容器中可能会出现问题，特别是当 Web 应用程序被加载到容器中和从容器中卸载时。当在 Web 容器中运行时，Log4j 2.6 不会使用 `ThreadLocal`，但是仍然会使用一些共享和缓存结构来帮助提高性能。

如果应用程序已经在使用旧版本的 Log4j，那你应该考虑直接将其升级到 2.6 版本并重新检查配置。通过使用可变参数（vararg），为传递给日志输出语句的参数创建一个临时数组，可以使 Log4j 减少分配的数量。如果是通过 SLF4J（The simple logging facade for Java，Java 简单日志门面）来使用 Log4j，这个门面（facade）仍然只支持两个参数。因此，要通过 SLF4J 使用更多参数，就不能使用无垃圾方法或重构代码库来直接使用 Log4j 2 库。

14.3　使用Real Logic库实现低延迟

Real Logic 是一家位于英国的公司，由 Martin Thompson 创立。Martin 以开创了机械共鸣的方法而闻名，该方法的基础就是理解底层细节如何影响高性能设计。Martin 对 Java 领域最著名的贡献之一是 Disruptor 模式。

> Martin 的博客也叫“机械共鸣”（mechanical sympathy），对于想提升其应用程序的性能极限的开发人员来说，这个博客和相关的邮件列表是非常好的学习资源。

Real Logic 公司的 GitHub 页面上有以下几个很受欢迎的开源项目，它们都借鉴了 Martin 和其他贡献者的专业经验。

Agrona
: 用于 Java 的高性能数据结构和工具方法。

Simple Binary Encoding (SBE)
: 一个高性能的编解码器。

Aeron
: 高效可靠的 UDP 单播、UDP 多播和 IPC 消息传输。

后面几节将探索这些项目，并研究使得这些库突破 Java 性能极限的设计理念。

Real Logic 公司还托管了一个利用这些库来实现的可迅速恢复的高性能 FIX 网关。不过本章不会进一步探讨。

14.3.1 Agrona

Project Agrona 这个名字源于威尔士和苏格兰的凯尔特神话，它是一个库，包含用于开发低延迟应用程序的构建块。在 12.3 节中，我们讨论过在适当的抽象层次上使用 `java.util.concurrent` 的想法，以避免重新发明轮子。

Agrona 为真正的低延迟应用程序提供了一组类似的库。如果已经证明标准库不能满足自己的用例，那么在编写自己的库之前，需要先评估这些库。这个项目进行了良好的单元测试，提供了不错的文档并拥有活跃的社区。

1. 缓冲区

Richard Warburton 写了一篇很好的文章来介绍缓冲区以及 Java 中的缓冲区存在的问题。

概括地说，Java 提供了一个 `ByteBuffer` 类，它为缓冲区（既可以是直接的，也可以是非直接的）提供了一个抽象。

直接缓冲区存在于通常所谓的 Java 堆之外，但仍在整个 JVM 进程的“C 堆”之内。因此，与堆内（非直接）缓冲区相比，它的分配率和回收率通常更低。直接缓冲区的优势是，JVM 会尝试在该结构上直接调用指令，而不需要中间映射。

`ByteBuffer` 的主要问题是，它是为通用场景设计的，因此无法应用特定于缓冲区类型的优化。例如，它们不支持原子操作，所以如果想在缓冲区上构建一个生产者 / 消费者风格的模型，就会出现问题。每次要包装一个不同的结构时，`ByteBuffer` 都会要求分配一个新的底层缓冲区。在 Agrona 中，复制可以避免，它支持 4 种带有不同特性的缓冲区，支持你定义和控制与每个缓冲区对象可能的交互。

- `DirectBuffer` 接口只提供了从缓冲区读取的能力，它位于接口层次结构的最上层。
- `MutableDirectBuffer` 接口扩展了 `DirectBuffer`，并向该缓冲区添加了写操作。
- `AtomicBuffer` 接口扩展了 `MutableDirectBuffer`，提供了排序行为。
- `UnsafeBuffer` 这个类使用 `Unsafe` 实现了 `AtomicBuffer`。

图 14-3 演示了 Agrona 的缓冲区类的继承层次结构。

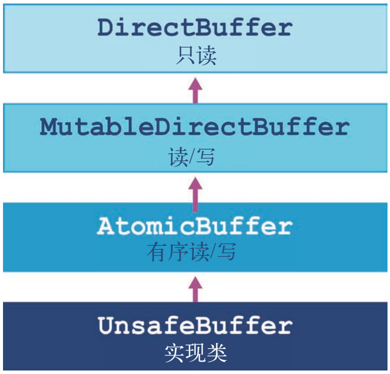


图 14-3：Agrona 缓冲区

正如你可能想象的，Agrona 中的代码非常底层，而且大量使用了 Unsafe，包括以下这样的代码片段：

```
// This horrible filth is to encourage the JVM to call memset()
// when address is even.
// TODO: check if this still applies when Java 9 is out!!!
UNSAFE.putByte(byteArray, indexOffset, value);
UNSAFE.setMemory(byteArray, indexOffset + 1, length - 1, value);
```

这并不是说 Agrona 会想尽各种办法进行特殊处理，恰恰相反，这段代码的需求是绕过 JVM 内部所应用的一个旧的优化，现在该优化已经是反优化了。这个类库深入到这种细节层次就是为了实现性能最大化。

Agrona 缓冲区支持通过各种 get 方法来访问底层数据，例如 getLong(int index)。尽管缓冲区是被包装起来的，但开发人员还是要知道数据的索引在什么位置。此外，put 操作支持将 long 类型的值放到缓冲区的某个特定位置。请注意，缓冲区并不是任何一种单一的类型，这取决于开发人员如何为其数据选择和管理恰当的结构。因为边界检查可以启用或禁用，所以死代码可以由 JIT 编译器优化掉。

2. 列表、映射和集

Agrona 提供了一系列列表实现，背后由 int 或 long 等基本类型的数组提供支持。正如我们在 11.1 节所提到的，Java 并没有提供在数组中并排摆放对象的布局机制，因此在标准的集合类中，得到的结果是由引用组成的数组。同样情况下，强制使用对象而不是基本类型，除了对象本身大小所带来的开销，还会导致自动装箱和拆箱。Agrona 还提供了用于 ArrayList 的工具类，支持从 ArrayList 中快速移除元素，但会破坏原有的顺序关系。[1]

Agrona 的映射和集实现将键和值并排存储在一个散列表结构中。如果键发生碰撞（collision），

注 1：这里说的工具类是指 ArrayListUtil，它所提供的用于移除元素的方法的处理逻辑是，移除指定位置的元素，并将最后一个元素放到该位置，而不是将指定位置之后的元素依次向左移动，所以会破坏原有的顺序关系。——译者注

那下一个值就会紧接着存储在散列表中原位置的后面。这种结构使其很适合快速访问存在于同一缓存行上的简单映射。

3. 队列

Agrona 有自己的并发包，其中包含有用的并发实用程序和结构，这包括队列和环形缓冲（ring buffer）区。

队列遵循标准的 `java.util.Queue` 接口，它们可以用来代替标准的队列实现。Agrona 队列还实现了 `org.agrona.concurrent.Pipe` 接口，该接口增加了对按顺序处理的容器的支持。特别是 `Pipe` 增加了对计数（count）、容量（capacity）和排干（drain）操作的支持，以便与该队列的消费者轻松交互。这些队列都是无锁的，并且为了适用于低延迟系统，它们还使用了 `Unsafe`。`org.agrona.concurrent.AbstractConcurrentArrayQueue` 为一系列队列提供了一级支持，这些队列将提供不同的生产者 / 消费者模型。这个 API 的一个有趣的部分是如下这些类：

```
/**
 * Pad out a cache line to the left of a producer fields
 * to prevent false sharing.
 */
class AbstractConcurrentArrayQueuePadding1
{
    @SuppressWarnings("unused")
    protected long p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12,
        p13, p14, p15;
}

/**
 * Values for the producer that are expected to be padded.
 */
class AbstractConcurrentArrayQueueProducer
        extends AbstractConcurrentArrayQueuePadding1
{
    protected volatile long tail;
    protected long headCache;
    protected volatile long sharedHeadCache;
}

/**
 * Pad out a cache line between the producer and consumer fields to prevent
 * false sharing.
 */
class AbstractConcurrentArrayQueuePadding2
        extends AbstractConcurrentArrayQueueProducer
{
    @SuppressWarnings("unused")
    protected long p16, p17, p18, p19, p20, p21, p22, p23, p24, p25,
        p26, p27, p28, p29, p30;
}

/**
 * Values for the consumer that are expected to be padded.
 */
```

```java
class AbstractConcurrentArrayQueueConsumer
        extends AbstractConcurrentArrayQueuePadding2
{
    protected volatile long head;
}

/**
 * Pad out a cache line between the producer and consumer fields to
 * prevent false sharing.
 */
class AbstractConcurrentArrayQueuePadding3
    extends AbstractConcurrentArrayQueuePadding2
{
    @SuppressWarnings("unused")
    protected long p31, p32, p33, p34, p35, p36, p37, p38, p39, p40,
        p41, p42, p43, p44, p45;
}

/**
 * Leftover immutable queue fields.
 */
public abstract class AbstractConcurrentArrayQueue<E>
    extends AbstractConcurrentArrayQueuePadding3
    implements QueuedPipe<E> {...}
```

> 值得注意的是，未来可能会使用 `sun.misc.contended`（或 `jdk.internal.vm.annotation.Contended`）来生成这种缓存行填充。

从 `AbstractConcurrentArrayQueue` 的代码片段可以看出，队列的内存布局有巧妙的（强制的）安排，以避免当该队列被消费者和生产者并发访问时发生伪共享。之所以需要这样填充，是因为 Java 和 JVM 并不保证字段在内存中的布局。

将生产者和消费者分别放在不同的缓存行上可以确保该结构能够在低延迟、高吞吐量的情况下正常执行。缓存行是用来访问内存的，如果生产者和消费者共用同一缓存行，那它被并发访问时就会出现问题。

这个 API 有以下 3 种具体实现，通过它们来实现分离，从而只有在代码中有必要的地方才需要进行协调。

`OneToOneConcurrentArrayQueue`

如果我们选择一个生产者和一个消费者，那就表明只有当生产者和消费者同时访问该结构时才会发生并发访问。我们只需要关注头部和尾部位置，因为这些位置每次只会被一个线程更新。

头部位置唯一会更新的情况是，通过轮询或排干操作从队列中获取元素；尾部位置唯一会更新的情况是 `put()` 操作。选择这种模式可以避免由于额外的协调检查而造成的不必要的性能损失，而其他两种类型的队列都需要这种检查。

`ManyToOneConcurrentArrayQueue`

另外，如果我们选择许多生产者，那就需要对尾部位置的更新进行额外的控制（因为这个位置可能已经被另一个生产者更新了）。在 `while` 循环中使用 `Unsafe.compareAndSwapLong`，直到尾部被更新，可以确保以无锁方式安全地更新队列尾部。此外，因为我们保证只有一个消费者，所以在消费者方面就不存在这样的竞争。

`ManyToManyConcurrentArrayQueue`

最后，如果我们选择许多生产者和消费者，那就需要对头部或尾部位置的更新进行协调控制。这个级别的协调和控制是通过一个包裹着 `compareAndSwap` 的 `while` 循环来实现的。在所有选择中，这里需要的协调是最多的，所以只有在需要这种级别的安全性时才会用到。

4. 环形缓冲区

Agrona 提供了 `org.agrona.concurrent.RingBuffer`，作为交换二进制编码消息的接口，用于进程间通信。 它使用一个 `DirectBuffer` 来管理堆外消息的存储。借助源代码中的一些 ASCII 技巧，我们可以看到消息被存储在一个 `RecordDescriptor` 结构中：

```
*   0                   1                   2                   3
*   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*  |R|                         Length                              |
*  +-+-------------------------------------------------------------+
*  |                            Type                               |
*  +---------------------------------------------------------------+
*  |                       Encoded Message                        ...
* ...                                                              |
*  +---------------------------------------------------------------+
```

Agrona 中实现了两种类型的环形缓冲区：`OneToOneRingBuffer` 和 `ManyToOneRingBuffer`。写操作接收一个源缓冲区，将消息写入当前缓冲区，而读操作是在一个 `onMessage` 风格的处理程序上回调。当许多生产者要在 `ManyToOneRingBuffer` 中写入时，可以调用 `Unsafe.storeFence()` 来手动控制内存同步。存储屏障“保证在这个屏障之前的所有写操作都已经完成”。

Agrona 有许多底层结构和有趣的方面。如果你想构建的软件系统也很注重细节，那这个项目就是一个很好的实验起点。

还有其他项目（如 JCTools）也提供了并发队列结构的变种。除非有非常特殊的使用场景要求，否则开发人员不应该忽视这些开源库的存在，应该避免重新发明轮子。

14.3.2 Simple Binary Encoding

SBE 是为了满足适合低延迟性能的二进制编码表示的需求而开发的，该编码专门为金融系统中使用的 FIX 协议创建。

> Simple Binary Encoding (SBE) 提供了与其他二进制编码不同的特性，并针对低延迟进行了优化。这种新的 FPL 二进制编码是对现有的唯一二进制编码（FAST）的补充，该编码于 2005 年开发，致力于降低市场数据的带宽利用率。
>
> ——Simple Binary Encoding 规范发布候选版本 1

SBE 与应用层有关，用于编码和解码消息，所使用的缓冲区来自 Agrona。SBE 经过优化，支持通过这些数据结构来传递低延迟消息而不会触发垃圾收集，同时优化内存访问等问题。它是专门为高频交易环境设计的，在这种环境下，往往必须在微秒或纳秒的时间内就对市场事件做出反应。

> 与使用 Google Protocol Buffers 和 ASN.1 来编码 FIX 的这种竞争方案相比，SBE 编码仅仅是在 FIX 协议组织内定义的。

高频交易的另一个关键的非功能性要求是操作应该始终保持快速。本书的一个作者曾亲眼看见过一个系统，它以相当高的吞吐量处理消息，但是突然因为一个垃圾收集错误暂停了两分钟。在低延迟应用程序中，这种暂停是完全不能接受的，完全避免垃圾收集是保证性能一致的方法之一。这些类型的性能问题可以通过浸泡测试（soak test）或其他长期运行的性能检测来识别。

低延迟应用程序的目标是尽可能地提高应用程序的性能。这可能会导致一场“军备竞赛”，在这场竞赛中，相互竞争的交易公司的团队试图在减少交易应用程序关键路径的延迟方面超越对方。

SBE 的作者提出了一系列的设计原则来反映这些问题，并解释了他们的想法。下面几节将探讨一些设计决策以及它们与低延迟系统设计的关系。

1. 免复制和原生类型映射

复制是有成本的，凡是用 C++ 做过编程的人可能都会时常被人发现不小心复制了对象。当对象很小的时候，复制成本并不高，但是随着对象大小的增加，复制的成本也会增加。

高级别的 Java 程序员可能并不总会考虑这个问题，因为他们已经习惯了使用引用和自动管理内存。SBE 中的免复制技术就是为了确保在编码或解码消息时不使用中间缓冲区而设计的。

然而，直接向底层缓冲区写入确实是有设计成本的，因为它不支持当前缓冲区无法容纳的较大的消息，如果要支持它们，就必须内置一个协议来分割消息并重新组合。

当进行免复制设计时，可以原生映射到合理的汇编指令的类型也会有所帮助。拥有一个与良好的汇编操作选择相对应的映射，可以极大地提高字段检索的性能。

2. 稳态分配

在 Java 中，分配对象自然会引入一个低延迟应用程序非常关注的问题。分配本身就需要 CPU 周期（即使是非常小的周期，比如一次 TLAB 分配），因此就会出现在使用完成后删除对象的问题。

垃圾收集往往是全部停顿的，即使是在大部分情况下进行并发工作的更高级的收集器也会引入一个暂停。虽然限制了绝对的暂停时间，但垃圾收集的过程也会在性能模型中引入一个有影响的变量。

似乎可以很自然地认为 C++ 能帮助解决这个问题，但是分配和回收机制也会引入问题。特别是，一些内存池可能会采用某种损害性能的加锁机制。

SBE 是免分配的，因为它在底层缓冲区之上使用了享元模式（flyweight pattern）。

3. 流式访问和字对齐访问

在 Java 中，通常我们无法控制对内存的访问。我们在 11.1 节曾讨论了 ObjectLayout，它建议以像 C++ 中向量那样的对齐方式来存储对象。通常 Java 中的数组都是引用的数组，因此无法在内存中按顺序读取。

SBE 被设计为以向前连续的方式对消息进行编码和解码，同时也会进行正确的组帧，以支持合理的字对齐。如果没有合理的字对齐，那在处理器层面就会开始出现性能问题。

4. 使用SBE

SBE 消息被定义为说明消息布局的 XML 模式文件。虽然现在 XML 普遍不受大家喜欢，但该模式确实为精确说明消息接口提供了一个很好的机制。XML 在 Eclipse 和 Intellij 等 IDE 中也有直接的工具链支持。

SBE 提供了一个命令行工具 `sbe-tool`，在给定一个模式后，它可以生成相应的编码器和解码器。其使用步骤如下：

```
# Fork or clone the project
git clone git@github.com:real-logic/simple-binary-encoding.git

# Build the project using your favorite build tool
gradle

# The sbe-tool will be created in
sbe-tool/build/libs

# Run the sbe-tool with a schema-a sample schema is provided at
# https://github.com/real-logic/simple-binary-encoding/blob/master/
    sbe-samples/src/main/resources/example-schema.xml
java -jar sbe-tool-1.7.5-SNAPSHOT-all.jar message-schema.xml

# When this command completes it will generate a series of .java files in the
    baseline directory
$ ls
BooleanType.java                GroupSizeEncodingEncoder.java
BoostType.java                  MessageHeaderDecoder.java
BoosterDecoder.java             MessageHeaderEncoder.java
BoosterEncoder.java             MetaAttribute.java
CarDecoder.java                 Model.java
CarEncoder.java                 OptionalExtrasDecoder.java
EngineDecoder.java              OptionalExtrasEncoder.java
EngineEncoder.java              VarStringEncodingDecoder.java
GroupSizeEncodingDecoder.java   VarStringEncodingEncoder.java
```

重要的是要记住，SBE 协议的核心部分之一是消息必须按顺序读取，也就是说本质上要按照模式中的定义来读取。关于如何开始使用这些消息，其教程可参阅 *SBE Java Users Guide*。

虽然这里演示的是在命令行中使用 SBE 工具，但更为常见的使用场景可能是将其集成到构建流水线中。

14.3.3　Aeron

Aeron 是我们要探讨的 Real Logic 软件栈中的最后一个产品。将其放在最后应该并不奇怪，因为它是建立在 SBE 和 Agrona 的基础之上的。Aeron 是一个为 Java 和 C++ 编写的 UDP（用户数据报协议）单播、多播和 IPC（进程间通信）消息传输。它与媒体层无关，因此可以很好地与 InfiniBand 一起工作。

本质上，这是一个通用的、包罗万象的消息传输协议，使用它可以让处于同一台机器上或者跨网络的应用程序通过 IPC 相互对话。Aeron 被设计为具有最大可能的吞吐量，旨在实现最低和最为可预测的延迟结果（一致性很重要，正如我们在 14.3.2 节所讨论的）。本节将探讨 Aeron API 以及一些设计决策。

1. 为什么要构建新的东西

之所以要构建像 Aeron 这样的一个新产品，主要原因之一是市场上的一些产品已经变得更加通用。这种情况并不是不好，而是当客户要求某些功能时（而且通常是为这些功能付费），就会把产品推向某个特定的方向，从而导致产品可能会变得臃肿，提供的功能也会比预想的多很多，甚至可能成为框架。

用底层的 Java 机制构建消息系统是非常有趣的，它可能是向公司或者社区内部的业余兴趣项目迈出的第一步。潜在的问题是，从低延迟的角度来看，对所需的功能缺乏经验会使这些兴趣项目很难应用，也很难为生产环境做好准备。确保产品从一开始就是为低延迟应用程序而构建的，同时又不牺牲其性能，仍然是一个艰难的挑战。

正如本章所强调的，Aeron 背后强大的设计原则是它是作为一个组件库而构建的，因此，你不需要受制于一个框架，如果只需要一个底层的数据结构，那么 Agrona 就会提供，不需要引入许多其他的依赖关系。

2. 发布者

在深入讨论 Aeron 之前，了解一下图 14-4 中描述的一些更高级别的组件是很有用的。

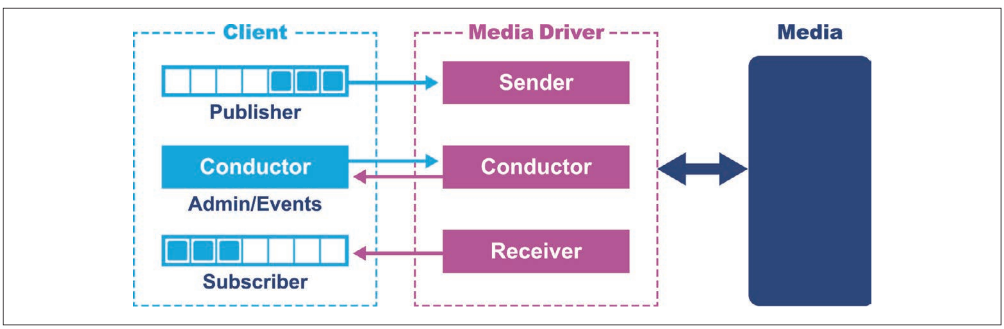


图 14-4：Aeron 主要组件的架构概览

确切的内容如下。

- **媒体**（media）是指 Aeron 所基于的通信机制，例如 UDP、IPC、InfiniBand 或其他媒介。重点是 Aeron 作为一个客户端可以从中抽象出来。
- **媒体驱动程序**（media driver）指的是媒体和 Aeron 之间的连接，支持提供配置并与该传输进行通信。
- **管理者**（conductor）负责管理，比如设置缓冲区以及监听新的订阅者和发布者的请求。它还将寻找否定应答（negative acknowledgment，NAK）并安排重传。这样发送者和接收者只需处理移动字节就可以使吞吐量最大化。
- **发送者**（sender）从生产者处读取数据并通过套接字发送出去。
- **接收者**（receiver）从套接字中读取数据，并将其转发到相应的通道和会话中。

媒体驱动程序通常是一个单独的进程，提供用于发送和接收消息的缓冲区。在不同的硬件部署上可以使用不同的媒体驱动程序进行优化，`MediaDriver.Context` 就是为媒体驱动程序设置优化的配置。媒体驱动程序也可以在同一个进程中以嵌入的方式启动，而被嵌入的进程则通过一个上下文或系统属性进行配置。要启动嵌入式媒体驱动程序，可以按照以下方式操作：

```
final MediaDriver driver = MediaDriver.launch();
```

Aeron 应用程序需要以发布者或订阅者的身份连接到媒体驱动程序，`Aeron` 类使之变得相当直接。Aeron 还有一个内部的 `Context` 类，可用于配置设置：

```
final Aeron.Context ctx = new Aeron.Context();
```

然后 Aeron 可以连接一个 `Publication`，通过给定的通道和流进行通信。因为 `Publication` 实现了 `AutoClosable` 接口，所以当 `try` 块执行完毕后，它将被自动清理：

```
try (Publication publication = aeron.addPublication(CHANNEL, STREAM_ID))
{...}
```

要发送一条消息，就要向发布者提供一个缓冲区，并通过提供操作（`offer()`）的结果来确定消息的状态。`Publication` 有一系列代表不同错误的 `long` 类型的常量，可以与 `offer()` 方法的结果进行比较：

```
final long result = publication.offer(BUFFER, 0, messageBytes.length);
```

发送消息就是这么简单，但要使其发挥作用，就应该让订阅者监听同一个媒体驱动程序。

3. 订阅者

订阅者的启动与发布者的类似，二者都需要访问媒体驱动程序，然后连接 Aeron 客户端。消费者的组件与图 14-4 所示的图互为镜像。消费者注册一个回调函数，收到消息时就会触发：

```
final FragmentHandler fragmentHandler =
    SamplesUtil.printStringMessage(STREAM_ID);

try (Aeron aeron = Aeron.connect(ctx);
     Subscription subscription = aeron.addSubscription(CHANNEL, STREAM_ID))
{
```

```
        SamplesUtil.subscriberLoop(fragmentHandler,
            FRAGMENT_COUNT_LIMIT, running).accept(subscription);
    }
```

这些示例只探讨了基本的设置，Aeron 项目中还有些示例探索了更高级的设置。

14.3.4 Aeron的设计

Martin Thompson 在 Strange Loop 的演讲中对 Aeron 进行了很好的介绍，同时说明了其构建的原因。本节将结合公开的文档，探讨视频中所讨论的一些内容。

1. 传输需求

Aeron 是用于消息的 OSI 第 4 层传输，因此它有一系列必须承担的责任。

排序（ordering）

因为从更底层传输中接收的数据包是乱序的，所以 Aeron 需要负责对乱序的消息进行重新排序。

可靠性（reliability）

当因数据丢失而出现问题时，Aeron 就必须提出重新传输数据的请求。在对旧数据的请求正在进行时，接收新数据的过程不应受到阻碍。这里的可靠性指的是连接级的可靠性，而不是会话级的可靠性（例如，多个进程重启时的容错能力）。

背压（back pressure）

订阅者在大流量的场景中将面临压力，所以服务必须支持流量控制和背压措施。

拥塞（congestion）

这是在网络饱和时会出现的问题，但如果正在构建一个低延迟应用程序，那它就不应该成为主要问题。Aeron 提供了一个激活拥塞控制的可选功能，使用低延迟网络的用户可以关闭它，对其他流量敏感的用户可以打开它。拥塞控制会影响在最佳路径中拥有足够网络容量的产品。

多路复用（multiplexing）

传输应该能在不影响整体性能的情况下，在一个通道上处理多个信息流。

2. 延迟和应用原则

Aeron 是在八大设计原则的驱动下设计的，它的维基页面简要介绍了这些原则。

在稳态运行情况下无垃圾收集（garbage-free in steady-state running）

垃圾收集暂停是造成 Java 应用程序中的延迟和不确定性的一个主要原因。Aeron 被设计为能够一直保持稳定状态以避免垃圾收集，因此它可以包含在同样遵守这一设计决策的应用程序中。

在消息路径下应用智能批处理（apply smart batching in the message path）

智能批处理（smart batching）是一个算法，为帮助处理收到突发消息时的情况而设计。在许多消息系统中，我们不能假定一天到晚都能以稳定的速度接收消息，大多数情况下会基于业务的事件，以突发的方式接收消息。如果在处理一条信息时接收到另一条信

息，那它也可以被捆绑到同一个网络包中，这取决于容量。通过使用适当的数据结构，Aeron 可以使这种批处理在不阻碍生产者对共享资源的写入的情况下运行。

在消息路径下的无锁算法（lock-free algorithms in the message path）
: 加锁会引入竞争，导致线程可能在其他线程运行时被阻塞，甚至在锁上停顿和唤醒都会使应用程序变慢。避免加锁可以防止由锁导致的速度下降。

在消息路径下的非阻塞 I/O（nonblocking I/O in the message path）
: 阻塞 I/O 会阻塞一个线程，而且唤醒的成本很高，但使用非阻塞 I/O 可以避免这些成本。

在消息路径下没有异常情况（no exceptional cases in the message path）
: 应用程序的大部分时间用于执行主要场景，而不是较小的边界场景。边界场景应该处理，但不能以牺牲主要场景的执行速度为代价。

应用单一写者原则（apply the single writer principle）
: 正如我们在 14.3.1 节讨论的 `ManyToOneConcurrentArrayQueue`，拥有多个写者就会涉及对队列访问进行高度的控制和协调，但是建立单一的写者就可以大大简化这个操作，并减少写入时的竞争。Aeron 的发布者是线程安全的，支持多个写者，但它的订阅者不是线程安全的，所以你想订阅的每个线程都需要一个写者。

首选非共享状态（prefer unshared state）
: 单一写者虽然解决了队列上的竞争问题，但它必须共享可变数据。在软件设计的各个环节中保持私有或本地状态都是首选，因为它大大简化了数据模型。

避免不必要的数据复制（avoid unnecessary data copy）
: 正如我们所提到的，虽然数据复制通常很便宜，但缓存行失效和逐出其他数据的可能性在 Java 和 C++ 中都会引发问题。尽量减少复制有助于防止这种意外搅动内存的现象。

3. Aeron底层是如何工作的

为了构建高效的消息处理系统，很多现有的协议引入了复杂的数据结构，如跳表（skip list）。主要因为指针的间接性，这种复杂性导致系统具有不确定的延迟特性。

> 从根本上说，Aeron 创建了一个复制的、持久的消息日志。
>
> ——Martin Thompson

通过 Aeron，可以用最清晰、最简单的方式在一个结构中构建消息序列。这初看上去虽然不是最为可能的选择，但 Aeron 大量使用了文件的概念。文件是可以跨感兴趣的进程共享的结构，通过使用 Linux 的内存映射文件功能，可以将所有对文件的调用导向内存，而不是物理文件的写入。

> Aeron 默认映射到 `tmpfs`（它是像文件一样挂载的易失性内存），其性能明显优于磁盘内存映射文件。

尾部指针用于记录最后一条消息的写入位置。图 14-5 演示了一条消息已经和它的头部指针一起被写入了当前的文件中。

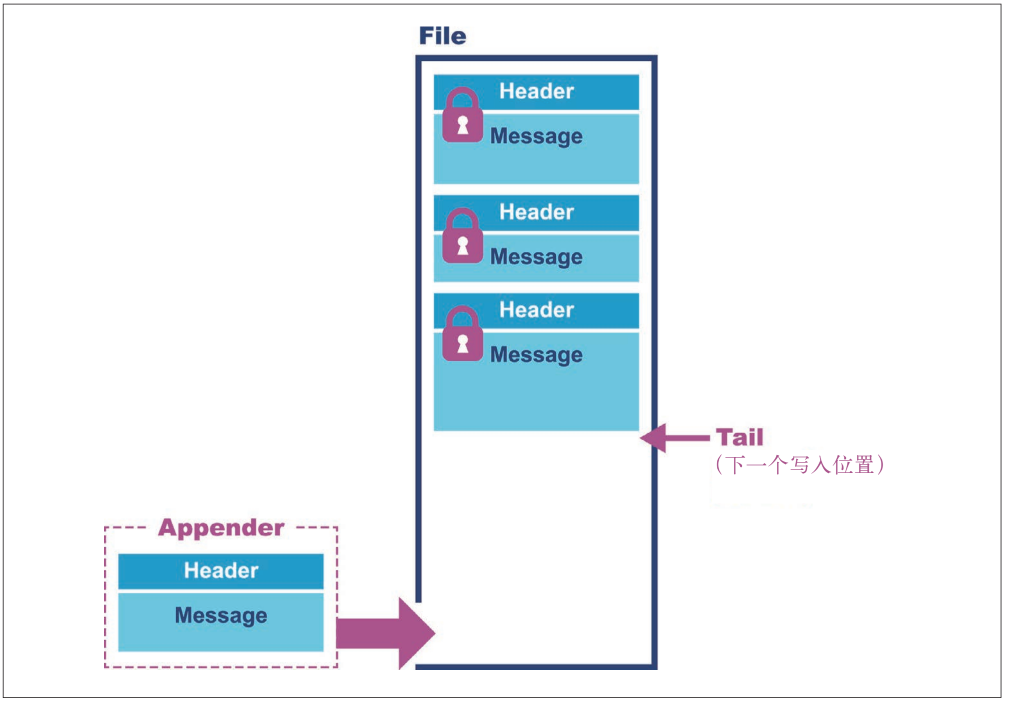
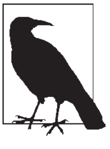


图 14-5：被写入文件的消息

这里的事件顺序相当有趣。尾部指针为文件中的消息保留了空间，它的增量操作是原子的，因此写者知道它所属部分的开始和结束位置。



使这个增量操作实现原子化的内部函数是在 Java 8 的关键补丁程序更新（critical patch update，CPU）中引入的。

这允许多个写者以无锁的方式更新文件，它也是高效建立文件写入协议的原因。但是我们怎么知道消息已经写完了呢？头部指针会在最后以原子方式写入文件来表明消息已经完成。

文件是持久性结构，写入后会增长，而且不会突变。从文件中读取记录不需要加锁，因为文件只能由一个观测进程打开和读取。但是否可以简单地拥有一个无限增长的文件呢？

这就会带来很多问题，因为它会在之前很好的内存映射文件中引入缺页错误和页面搅动，但是可以通过 3 个文件来解决这些问题：active、dirty 和 clean。active 代表当前正在写入的文件，dirty 是之前写入的文件，clean 是下一个要写入的文件。这些文件是轮流使用的，以避免出现由大文件造成的延迟。

消息是绝对不允许跨文件的。如果消息尾部超出了 active 文件的末端，那插入进程会填充该文件的剩余部分，并将消息写入 clean 文件。 从 dirty 文件中可以永久地将事务日志存档（archive）并深度存储。

处理缺失消息的机制也非常灵活，避免了前面提到的跳表和其他结构。消息的头部包含了它的排序信息，当一个消息被插入时，如果顺序不对，就会给前面的消息留下一个空位。当缺失的消息到达时，它就可以被插入到文件中的正确位置。这样就可以得到一系列不断增加的消息，中间不会有空隙或其他结构。从机械共鸣的角度看，将数据以递增方式排序的一个好处是处理起来非常快速。

水位标志（watermark）代表最后收到消息时的那个位置。如果水位标志和尾部指针在一段时间内不一样，就说明有消息缺失。为了解决消息缺失，系统会发送 NAK 来请求它们。可以为一条消息发送一个 NAK，一旦收到消息，NAK 就会被填充。

这个协议有个有趣的副作用，即基于 `streamId`、`sessionId`、`termId` 和 `termOffset` 所收到的每条消息都有一个独特的方式来识别消息中的字节。我们可以使用 Aeron Archive 来录制和重放消息流，并且通过结合存档和这种独特的表示方式，可以唯一地识别历史上的所有消息。

日志文件是 Aeron 能够保持速度和状态的核心，其简单优雅的设计，使 Aeron 能够与成熟的（昂贵的）多播产品媲美，在很多情况下甚至能击败它们。

14.4　小结

日志记录是所有生产级应用程序中不可或缺的一部分，所使用的日志记录器的类型对整体的应用程序性能有非常大的影响。当涉及日志记录时，重要的是要将应用程序当作一个整体而不只是执行日志记录语句来考虑，同时注意它对其他 JVM 子系统（如线程使用和垃圾收集）的影响。

本章包含一些低延迟库的简单例子，从最底层开始，一直到一个完整的消息系统实现。显然，应该把低延迟系统的目的和目标应用于整个软件栈，从最底层的队列一直到更高层次的应用。低延迟、高吞吐量的系统需要大量的思考、经验和控制，这里讨论的许多开源项目是基于大量丰富的经验构建起来的。如果你需要创建一个新的低延迟系统，只要能从底层设计目标一直坚持到顶层应用程序，这些项目都将为你节省几天甚至几周的开发时间。

本章开始提出的一个问题是 Java 和 JVM 可以在多大程度上应用于高吞吐量的应用程序。使用任何语言编写低延迟、高吞吐量的应用程序都是非常困难的，但是在所有可用的语言中，Java 提供了最好的工具和生产效率。此外，Java 和 JVM 确实增加了另一个抽象层次，我们需要对其进行管理，并在某些情况下加以规避。同时，考虑硬件、JVM 性能和更底层的问题也非常重要。

我们在日常的 Java 开发工作中一般不会遇到这些较为底层的问题。使用本章所讨论的较新的免分配的日志库、数据结构和消息传递协议，可以大大降低进入门槛，因为很多复杂的问题已经被开源社区解决了。

第 15 章

Java 9以及Java的未来方向

在编写本书时，Java 9 还处于积极开发中。新版本包含很多与性能相关的特性，以及与 Java/JVM 应用程序工程师相关的增强。

本章第一部分研究 Java 9 中性能工程师应该了解的有关平台新增和改动的地方。

事实上，对于大多数开发人员而言，Java 9 确实是由“模块和其他一切”组成的。正如 Java 8 是关于 Lambda 及其相关部分（流、默认方法以及有关函数式编程的一些小的方面）的，所以可以认为 Java 9 主要是关于模块的。

模块是一种构建和部署软件的新方式，不适合采用零碎敲打的方式进行处理。虽然它们代表了一种构建架构良好的应用程序的现代方式，但无论如何，团队和项目可能需要一段时间才能看到采用模块的长期效益。就目的而言，模块并不具有任何实际性能上的意义，因此我们不会尝试对其进行任何详细的讨论，而是将重点放在较小但与性能相关的变化上。

对 Java 9 模块感兴趣的读者，应当查阅相关的参考资料，比如 Sander Mak 和 Paul Bakker 合著的《Java 9 模块化开发》。

本章的大部分内容是对未来的讨论，因为在编写本书时，未来的这些方向已经存在了。Java 平台生态系统有一些正在进行的项目，这些项目有可能从根本上改变 JVM 应用程序的性能格局。作为总结，本书最后将介绍一下这些项目，并了解它们与 Java 性能专业人员的关系。

15.1　Java 9中小的性能增强

本节将探讨 Java 9 中与性能相关的增强。虽然其中有些增强非常小，但对某些应用程序来说可能意义重大。我们将重点讨论如下这些变化：

- 分段式代码缓存
- 紧凑的字符串
- 新的字符串连接
- C2 编译器的改进
- G1 的变化

15.1.1　分段式代码缓存

Java 9 带来的一个改进是将代码缓存分割成单独的区域，分别用于：

- 非方法代码，如解释器；
- 剖析过的代码（来自客户端编译器的级别 2 和级别 3 代码）；
- 未剖析的代码（级别 1 和级别 4）。

这应该可以减少清扫时间（非方法区域不需要清扫），对于完全优化的代码也能有更好的局部性。分段式代码缓存的缺点是在其他区域仍有空间的情况下，一个区域可能会被填满。

15.1.2　紧凑的字符串

在 Java 中，字符串的内容总是以 `char[]` 的形式存储。由于 `char` 是一个 16 位的类型，因此用于存储 ASCII 字符串的空间大约是实际所需要的两倍。Java 平台一直把这种开销作为简化 Unicode 处理而值得付出的代价。

Java 9 提供了**紧凑的字符串**，可以按字符串进行优化。如果这个字符串可以用 Latin-1 表示，那么它会被表示为一个字节数组（其中的字节被理解为对应的 Latin-1 字符），这就节省了用 `char` 表示时无意义的、内容为零的字节。在 Java 9 `String` 类的源代码中，变化看起来是这样的：

```
private final byte[] value;

/**
 * The identifier of the encoding used to encode the bytes in
 * {@code value}. The supported values in this implementation are
 *
 * LATIN1
 * UTF16
 *
 * @implNote This field is trusted by the VM, and is a subject to
 * constant folding if String instance is constant. Overwriting this
 * field after construction will cause problems.
 */
private final byte coder;
```

```
    static final byte LATIN1 = 0;
    static final byte UTF16 = 1;
```

在 Java 9 中，`value` 字段现在是 `byte[]`，而不是像早期版本中的 `char[]`。

> 可以通过传递 `-XX:-CompactStrings` 或 `-XX:+CompactStrings`（默认值）来关闭或打开该功能。

对于那些堆非常大，而且其中包含大量仅为 Latin-1（或 ASCII）的字符串数据的应用程序，比如 ElasticSearch、缓存和其他相关组件，这一变化的影响最为显著。对于这些应用程序来说，仅仅因为这个改进，可能就值得迁移到 Java 9 运行时。

15.1.3　新的字符串连接

考虑这段简单的 Java 代码：

```
public class Concat {
  public static void main(String[] args) {
    String s = "("+ args[0] + " : "+ args[1] +")";
    System.out.println(s);
  }
}
```

从 Java 5 开始，这个语言特性在去掉语法糖之后，就变成了一系列涉及 `StringBuilder` 类型的方法调用。这会生成相当多的字节码。

```
public static void main(java.lang.String[]);
  Code:
     0: new           #2      // class java/lang/StringBuilder
     3: dup
     4: invokespecial #3      // Method java/lang/StringBuilder."<init>":()V
     7: ldc           #4      // String (
     9: invokevirtual #5      // Method java/lang/StringBuilder.append:
                              // (Ljava/lang/String;)Ljava/lang/StringBuilder;
    12: aload_0
    13: iconst_0
    14: aaload
    15: invokevirtual #5      // Method java/lang/StringBuilder.append:
                              // (Ljava/lang/String;)Ljava/lang/StringBuilder;
    18: ldc           #6      // String  :
    20: invokevirtual #5      // Method java/lang/StringBuilder.append:
                              // (Ljava/lang/String;)Ljava/lang/StringBuilder;
    23: aload_0
    24: iconst_1
    25: aaload
    26: invokevirtual #5      // Method java/lang/StringBuilder.append:
                              // (Ljava/lang/String;)Ljava/lang/StringBuilder;
    29: ldc           #7      // String )
    31: invokevirtual #5      // Method java/lang/StringBuilder.append:
                              // (Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
34: invokevirtual #8      // Method java/lang/StringBuilder.toString:
                          // ()Ljava/lang/String;
37: astore_1
38: getstatic     #9      // Field java/lang/System.out:
                          // Ljava/io/PrintStream;
41: aload_1
42: invokevirtual #10     // Method java/io/PrintStream.println:
                          // (Ljava/lang/String;)V
45: return
```

然而在 Java 9 下，编译器生成了完全不同的字节码：

```
public static void main(java.lang.String[]);
  Code:
     0: aload_0
     1: iconst_0
     2: aaload
     3: aload_0
     4: iconst_1
     5: aaload
     6: invokedynamic #2,  0  // InvokeDynamic #0:makeConcatWithConstants:
                              // (Ljava/lang/String;Ljava/lang/String;)
                              // Ljava/lang/String;
    11: astore_1
    12: getstatic     #3      // Field java/lang/System.out:
                              // Ljava/io/PrintStream;
    15: aload_1
    16: invokevirtual #4      // Method java/io/PrintStream.println:
                              // (Ljava/lang/String;)V
    19: return
```

这依赖于 `invokedynamic`，我们曾在 9.1.1 节中提过。通过查看 `javap` 命令 `verbose` 选项的输出，可以看到常量池中的引导（bootstrap）方法。

```
0: #17 REF_invokeStatic java/lang/invoke/StringConcatFactory.
    makeConcatWithConstants:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/
    lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/String;
    [Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
```

这使用了 `StringConcatFactory` 中的一个名为 `makeConcatWithConstants()` 的工厂方法来产生一个用于字符串连接的具体手段。这种技术可以使用许多不同的策略，包括为一个新的自定义方法编写字节码。它在某些方面类似于使用预处理语句（prepared statement）来执行 SQL，而不是简单的字符串装配。

这个小的变化预计不会对许多应用程序的整体性能产生很大的影响。然而这一变化确实表明 `invokedynamic` 有更广泛的应用，也说明了平台发展的大致方向。

15.1.4　C2编译器的改进

C2 编译器现在已经相当成熟，而且人们（像 Twitter 等公司，甚至像 Cliff Click 等专家）普遍认为，在当前的设计中不太可能再有大的改进，这意味着任何改进必然都是边边角角的。然而，使用现代 CPU 上的**单指令多数据流**（SIMD）扩展有可能带来更好的性能。

与其他编程环境相比，由于以下这些平台特性，Java 和 JVM 在挖掘这些潜能方面处于有利地位：

- 字节码是平台无关的；
- JVM 在启动时执行 CPU 探测，所以它知道硬件在运行时的执行能力；
- JIT 编译是动态生成代码，所以它可以使用主机上所有可用的指令。

正如 10.7 节所讨论的，这些改进需要靠虚拟机内部函数来实现。

> 如果 HotSpot 虚拟机用手写的汇编代码或手写的编译器中间语言（一种编译器内部机制）来代替被注解的方法以提高性能，就可以说该方法被内部化了。
>
> ——@HotSpotIntrinsicCandidate JavaDoc

HotSpot 已经支持一些 x86 SIMD 指令，包括：

- Java 代码的自动向量化；
- C2 中的超字优化（superword optimization）能从顺序代码中推导出 SIMD 代码；
- JVM SIMD 内部函数，包括数组复制、填充和比较。

Java 9 版本包含了一些已修复的问题，为了更充分地利用 SIMD 和相关处理器特性，这些问题改进了内部函数或者引入了新的内部函数。从发布说明可以看出，因为加入了增强版的内部函数，所以下列问题已经修复并关闭：

- 循环后部标记向量；
- 超字循环展开分析；
- 多版本化的边界检查消除；
- 支持向量化双精度的 `sqrt`；
- 改进的并行流的向量化；
- 超字增强，以支持 Intel AVX CPU 上的向量条件移动（CMovVD）。

总的来说，应该将内部函数视为具体点上的修补，而不是通用技术。其优势是功能强大、轻量级以及灵活，但因为要支持多个不同的架构，所以潜在的开发和维护成本很高。SIMD 技术是有用且受欢迎的，但很显然，这种方法给性能工程师带来的回报只会越来越低。

15.1.5 新版G1收集器

正如 7.4 节所讨论的，G1 的设计是为了同时解决几个问题，它提供了一些功能，比如更易于调优和更好地控制停顿时间。在 Java 9 中，它成为默认的垃圾收集器。这意味着从 Java 8 迁移到 Java 9 的应用程序，就算没有明确选择收集器，垃圾收集算法也会改变。不仅如此，Java 9 中的 G1 版本也与 Java 8 中的不同。

Oracle 声称，在其基准测试中，新版本的性能大大优于 Java 8 中的现有版本。然而，这并没有得到任何公开发表的结果或研究的支持，目前我们最多只有坊间证据。

希望大多数应用程序不会受到这种算法变化带来的负面影响。然而，如果原来使用的是 Java 8 的默认收集器或 G1，而在迁移到 Java 9 之后，性能受到了影响，那么所有这些应用程序都应该做一个完整的性能测试。

15.2 Java 10和未来版本

在编写本书时，Java 9 刚刚发布。因而 Java 平台的开发工作现已完全转向了下一个版本，即 Java 10。本节首先讨论将在下一个版本中生效的新发布模式，然后再介绍目前已知的 Java 10 的情况。

15.2.1 新的发布流程

就在 Java 9 发布的前几天，Oracle 宣布了一个全新的 Java 发布模式，该模式将从 Java 10 开始进行。过去 Java 新版本的发布都是由功能驱动的，针对特定版本进行了大的改动，必要时会推迟发布直到重要特性准备就绪。这种发布方式导致 Java 9 的发布时间严重推迟，同时也使 Java 8 受到了影响。

特性驱动的发布周期也对 Java 平台的整体开发速度产生了更深层次的影响。由于制作和全面测试一个版本所需要的周期较长，因此关键特性实际上会阻碍其他较小特性的发布。在开发周期临近结束时发布一个版本，这意味着源码库极有可能在可用的开发周期中处于锁定或半锁定的状态。

从 Java 10 开始，该项目已经转向了严格的基于时间的模式。每 6 个月就会发布一个包含新特性的 Java 新版本。这些版本被称为**特性版本**，相当于原有模式中的主要版本。

> 特性版本通常不会像原来的主要版本那样包含很多的新功能或新变化。但无论如何，主要功能有时仍将在某个特性版本中出现。

Oracle 还将为一些特性版本提供**长期支持**（long-term support，LTS）版本，并只用这样的版本作为其专有 JDK。所有其他版本都将是 OpenJDK 二进制版本，使用 GNU 公共许可证（GPL）类路径除外。开源的 Java 构建一直以来都使用该许可。其他厂商可能支持他们自己的二进制版本，也可能支持 LTS 以外的版本。

15.2.2 Java 10

在编写本书时，Java 10 的范围还没有完全确认并锁定。因此，从现在到发布之前，其范围和内容仍有可能发生重大变化。例如，在 Java 9 发布后的几周内，人们就曾公开讨论过未来将使用的版本编号机制。

新的 JVM 特性或增强都是通过 Java 增强过程（Java enhancement process）来进行跟踪的。每个 JDK 增强提案（JDK Enhancement Proposal，JEP）都有一个方便追踪的编号。如下是将作为 Java 10 的一部分所发布的主要特性，并非所有的特性都与性能有关，甚至有些不直接面向开发人员：

- 286：局部变量类型推导；
- 296：将 JDK 源代码从多个仓库（forest）合并到一个仓库；
- 304：垃圾收集器接口；

- 307：G1 的并行 Full GC；
- 310：应用程序类数据共享；
- 312：线程局部握手。

JEP 286 允许开发人员在局部变量声明中减少一些样本代码，如下代码在 Java 中已经是合法的：

```
var list = new ArrayList<String>();  // 推导ArrayList<String>
var stream = list.stream();          // 推导Stream<String>
```

这种语法只能用于带初始化的局部变量，以及 for 循环中的局部变量。当然，这完全是在源码编译器中实现的，对字节码或性能没有实际影响。尽管如此，对这一变化的讨论和反应还是说明了语言设计的一个重要方面，即以函数式程序员和计算机科学家 Philip Wadler 的名字命名的 **Wadler 定律**：

> 对一种语言特性争论的情绪激烈程度会随着以下几个方面的变化而增加：语义、句法、词法以及注释。

在其他的变化中，JEP 296 纯粹是 JDK 源代码的整理，JEP 304 增加了不同垃圾收集器的代码隔离，并在 JDK 构建中为垃圾收集器引入了一个清晰的接口。这两个变化对性能都没有任何影响。

剩下的 3 个变化或多或少都会对性能产生一定的影响。JEP 307 解决了 G1 垃圾收集器之前没解决的一个问题，如果它不得不回退到 Full GC，那么性能会变得非常糟糕。从 Java 9 开始，当前 G1 的 Full GC 的实现使用的是单线程（即串行）标记 – 清除 – 压缩算法。JEP 307 的目的是将该算法并行化，以便在 G1 的 Full GC 万一发生时，有与并发收集相同数量的线程可用。

JEP 310 扩展了 Java 5 引入的一个叫作**类数据共享**（class-data sharing，CDS）的功能。其理念是，JVM 记录一组类，并将其处理成一个共享的归档文件。在下一次运行时，可以通过内存映射加载该文件来减少启动时间。当多个 JVM 在同一主机上运行时，它还可以跨 JVM 共享，从而减少整体的内存占用。

到 Java 9 为止，CDS 还只允许启动类加载器加载归档文件。这个 JEP 的目的是扩展这种行为以允许应用程序和自定义类加载器使用归档文件。该特性确实存在，但目前只能在 Oracle JDK 中使用，而不能在 OpenJDK 中使用。因此，这个 JEP 本质上是将该特性从私有的 Oracle 源码中转移到开放的仓库中。

最后，JEP 312 为提高虚拟机性能奠定了基础，它使得在应用程序线程上执行回调而无须执行全局虚拟机安全点成为可能。这意味着 JVM 可以停止单个线程，而不是所有线程。这一改变将使如下这些改进成为可能：

- 减少获取栈轨迹采样的影响；
- 通过减少对信号的依赖性，实现更好的栈轨迹采样；
- 在撤销偏向锁时，可以仅停掉个别线程来改进偏向锁；
- 从 JVM 中移除一些内存屏障。

总的来说，Java 10 不太可能再有任何大的性能改进，相反，它代表的是新的、更频繁的和渐进式发布周期的第一个版本。

15.3 Java 9及更高版本中的Unsafe

任何关于 Java 未来的讨论，必须提到有关 `sun.misc.Unsafe` 类及相关副作用的争议。正如在 12.3 节中看到的，虽然 `Unsafe` 是一个内部的类，并不是标准 API 的一部分，但在 Java 8 中已经成为一个事实上的标准。

从库开发人员的角度来看，`Unsafe` 包含了一系列安全程度不同的功能。虽然用于访问 CAS 硬件的方法基本上是安全的，但不是标准的。而其他的方法则是不安全的，包括相当于指针算术之类的方法。但是，有些“完全不安全”的功能无法通过其他任何方式获得。Oracle 将这些功能称为**关键的内部 API**，在相关 JEP 中已有讨论。

主要的问题是，如果 `sun.misc.Unsafe` 及相关类中的某些特性没有替代方案，那主要的框架和库都将无法继续正常工作，这进而又会间接影响到每个使用各种框架的应用程序，并且在现代环境中，这基本上也会影响生态系统中的每个应用程序。

Java 9 添加了 `--illegal-access` 运行时开关，以控制这些 API 在运行时的可访问性。这些关键的内部 API 计划在未来的版本中通过替代方案来支持，但在 Java 9 发布之前不可能实现这一计划。因此，必须维护对以下类的访问：

- `sun.misc.{Signal,SignalHandler}`
- `sun.misc.Unsafe`
- `sun.reflect.Reflection::getCallerClass(int)`
- `sun.reflect.ReflectionFactory`
- `com.sun.nio.file.{ExtendedCopyOption,ExtendedOpenOption, Extended WatchEventModifier,SensitivityWatchEventModifier}`

在 Java 9 中，这些 API 被定义在 JDK 特有的模块 `jdk.unsupported` 中，并通过该模块导出，其声明是这样的：

```
module jdk.unsupported {
    exports sun.misc;
    exports sun.reflect;
    exports com.sun.nio.file;

    opens sun.misc;
    opens sun.reflect;
}
```

尽管 Oracle 提供了这种临时性支持（还是相当勉强的），但很多框架和库在向 Java 9 迁移时还是会遇到问题，而且 Oracle 也没有宣布对关键内部 API 的这种临时性支持何时失效。

话虽如此，我们在创建这些 API 的替代品方面已经取得了明显进展。例如，在 JEP 259 中定义的栈遍历 API 可以使用 `getCallerClass()` 功能。另外还有一个非常重要的新 API，其目的是开始取代来自 `Unsafe` 中的特性，接下来将看到。

Java 9中的VarHandles

我们已经在前面章节介绍了方法句柄和 `Unsafe`。

方法句柄提供了一种方式，可以直接操作指向方法的可执行引用，但最初的功能对字段来说并不全面，因为它只提供了 getter 和 setter 的访问方式。这还不够，因为 Java 平台提供的数据访问模式超出了这些简单用例。

在 Java 9 中，方法句柄已经被扩展到包括在 JEP 193 中定义的**变量句柄**（variable handle）。该提案的目标之一是填补上面提到的空白，并且在这个过程中，为 `Unsafe` 中的 API 提供安全的替代品。具体的替代品包括 CAS 功能和对 `volatile` 字段和数组的访问。另一个目标是作为 JMM 更新的一部分，允许对 JDK 9 中可用的内存顺序模式进行低级访问。

接下来快速看一个演示如何替换 `Unsafe` 中的方法的简单示例：

```
public class AtomicIntegerWithVarHandles extends Number {

    private volatile int value = 0;
    private static final VarHandle V;

    static {
        try {
            MethodHandles.Lookup l = MethodHandles.lookup();
            V = l.findVarHandle(AtomicIntegerWithVarHandles.class, "value",
                int.class);
        } catch (ReflectiveOperationException e) {
            throw new Error(e);
        }
    }

    public final int getAndSet(int newValue) {
        int v;
        do {
            v = (int)V.getVolatile(this);
        } while (!V.compareAndSet(this, v, newValue));

        return v;
    }
    // ...
```

这段代码示例基本上等同于我们在 12.3 节看到的原子整数的示例，该示例演示了如何用 `VarHandle` 替代不安全技术的使用。

在编写本书时，由于循环依赖，实际的 `AtomicInteger` 类还没有被迁移到使用 `VarHandle` 机制，仍然依赖于 `Unsafe`。尽管如此，Oracle 强烈建议所有的库和框架尽快迁移到新的支持机制。

15.4　Valhalla项目和值类型

Valhalla 项目的使命是成为“一个探索和孵化高级的 Java 虚拟机和语言候选特性的试验场”。该项目的主要目标如下：

- 使 JVM 内存布局行为与现代硬件的成本模型相匹配；
- 扩展泛型以支持对所有类型进行抽象，包括基本类型、值，甚至 void；
- 使现有的库（特别是 JDK）能在保持兼容性的前提下持续演进，以充分利用这些特性。

这段描述提到了该项目最引人注目的工作之一：在 JVM 中探索**值类型**的可能性。

回想一下，直到第 9 个版本（包括版本 9），Java 都只有两种类型的值：基本类型和对象引用。换句话说，Java 环境有意没有提供对内存布局的底层控制。作为特例，这意味着 Java 没有像结构体（struct）这样的东西，任何复合数据类型都只能通过引用来访问。

为了理解这种做法的影响，来看看数组的内存布局。图 15-1 显示了一个由基本类型 int 组成的数组。由于这些值不是对象，因此它们被安排在相邻的内存位置上。



图 15-1：int 数组

相比之下，因为装箱的整型数（即 Integer）是一个对象，所以要通过引用来查找。这就意味着一个 Integer 对象的数组将是一个引用的数组，如图 15-2 所示。

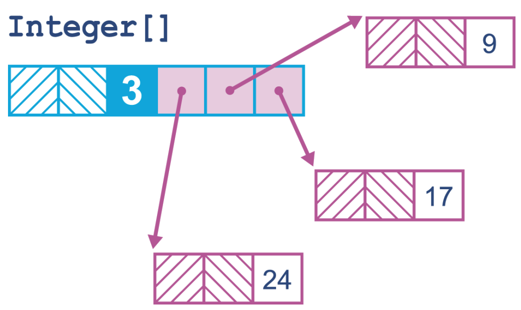


图 15-2：Integer 数组

20 多年来，Java 平台一直是按照这种内存布局模式来运作的。其优势是简单，但代价是性能上的损失：处理对象的数组不可避免有间接访问，以及随之而来的缓存未命中。

因此，许多关注性能的程序员希望能够定义内存布局更高效的类型。这还包括为消除每个复合数据项需要一个完整对象头所带来的开销。

例如，三维空间中的一个点用 Point3D 表示，实际上只包含了 3 个空间坐标。一直到 Java 9，这样的类型都只能用一个具有 3 个字段的对象类型来表示。

```
public final class Point3D {
    private final double x;
    private final double y;
```

```
        private final double z;

        public Point3D(double a, double b, double c) {
            x = a;
            y = b;
            c = z;
        }
        // getter方法和其他样板代码略
    }
```

因此，一个点的数组的内存布局将如图 15-3 所示。

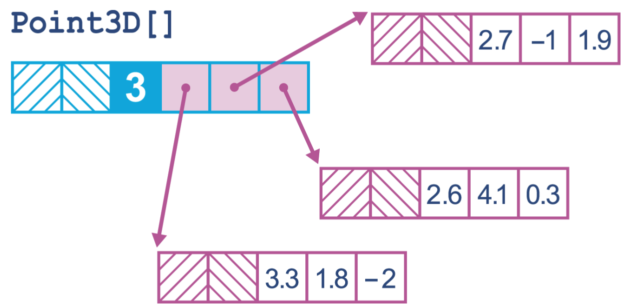


图 15-3：Point3D 数组

在处理这个数组时，每个条目必须通过一次额外的间接访问才能获得每个点的坐标。对于数组中的每个点，这有可能引发一次缓存未命中，而不能带来任何好处。

此外，对象的同一性（identity）对于 Point3D 类型来说没有意义。这意味着只有当对象的字段都相等时，它们才是相等的。显然，这就是所谓的 Java 生态系统中**值类型**的含义。

如果这个概念可以在 JVM 中实现，那么对于空间点这样的简单类型，图 15-4 所示的内存布局（实际上是一个 struct 数组）将更高效。



图 15-4：“类结构体”Point3D 的数组

不仅如此，随后还出现了其他的可能性，比如用户定义的类型，其行为方式与内置的基本类型类似。

然而，这个领域还存在着一些关键的概念上的问题，其中有一个与 Java 5 加入泛型时所做的设计决策有关。事实上 Java 类型系统缺乏一个**顶层类型**，所以 Object 和 int 没有一个共同的父类。也可以说，Java 的类型系统不是**单根类型**。

其结果是，Java 的泛型只能用于引用类型（`Object` 的子类型），没有办法为基本类型构造一个同样意义的东西，比如 `List<int>`。Java 使用**类型擦除**（type erasure）来实现向后兼容的泛型类型而不是引用类型，虽然这种机制饱受诟病，但缺乏顶层类型，进而导致缺乏基本类型的集合，并不是该机制的责任。

如果要扩展 Java 平台以引入值类型，那么自然会出现值类型是否可以用作类型参数值的问题。如果不能，这似乎会极大限制其实用性。因此，值类型的设计总是包含这样一个假设：它们作为增强形式的泛型中的类型参数值是有效的。

在编写本书时，该思路带来了这样的设计，即 JVM 的类和接口类型应该有以下 3 种不同的**类型**：

- 引用类型（`R`），表示对一个类的实例的引用，它有一个实际的对象，或者为 `null`；
- 值类型（`Q`），是缺乏同一性的**值类**的实例；
- 通用类型（`U`），可以是 `R` 或 `Q`。

这就引出了一个问题：应该如何理解现有的类文件中的类型信息？ 也就是说，现有的 `L` 类型（对应于 Java 9 类文件中的当前类型）应该是 `R` 类型、还是 `U` 类型，抑或是简单当作我们从未见过的 `Q` 类型？

出于兼容性考虑，同时也是为了让我们能够将泛型的定义扩展到包含 `Q` 类型，`L` 类型会被理解为 `U` 类型，而不是 `R` 类型。

这是一个相当早期的原型，仍然有很多设计问题需要解决，比如，值类型在虚拟机级别上是否需要宽度可变的值。

在 Java 9 中，在虚拟机级别上的所有类型都是宽度固定的值。因为基本类型是 1、2、4 或 8 字节宽，对象引用是指针，所以它们是 1 个机器字宽。在现代硬件上，这意味着根据机器硬件架构的不同，引用可以是 32 位或 64 位。

增加值类型是否意味着字节码需要容纳宽度可变的类型？在字节码中是否还有空间来容纳所需要的指令？就目前来看，只需要增加以下两个新的操作码：

- `vdefault`，生成值类的实例，默认为 `Q` 类型；
- `withfield`，生成一个新的输入类型的值，并在遇到输入为 `null` 或者非法值时抛出异常。

有些字节码需要改造以处理新的 `Q` 类型，并且在虚拟机级别上也需要做大量的工作，从而使核心库在保持兼容的前提下继续演进。

> Valhalla 项目可能是基于性能方面的考虑来推进的，但更好的方法是在**不**放弃性能的前提下，增强抽象、封装、安全、表现力和可维护性。
>
> ——Brian Goetz

由于发布时间表的变化，目前还不清楚 Java 的哪个版本最终会将值类型作为产品特性引入进来。作者最好的猜测是，它们会在 2019 年的某个版本中出现，但这尚未得到 Oracle 的证实。

15.5 Graal和Truffle

HotSpot 中的 C2 编译器已经取得了巨大的成功。不过近年来它的回报率明显下降，并且这款编译器在过去几年没有什么大的改进。无论从哪个角度来说，C2 都已经走到了生命周期的尽头，现在必须被取代了。

目前想要交付的新产品的研究是围绕 Graal 和 Truffle 展开的。前者是一个专门的 JIT 编译器，后者是一个解释器生成器，用于支持 JVM 运行时上的各种语言。

JIT 编译器的一个潜在的改进途径是，C2 编译器是用 C++ 编写的，这就暴露了一些潜在的严重问题。当然，因为 C++ 是一种使用手动内存管理的不安全的语言，所以 C2 编译器代码中的错误会导致虚拟机崩溃。不仅如此，由于 C2 中的代码一直在反复修改和迭代，所以代码很难得到维护和扩展。

为了提升，Graal 正在尝试另一种方法，它是一个用 Java 为 JVM 编写的 JIT 编译器。JVM 用来与 Graal 对话的接口称为 JVM 编译器接口（JVMCI），并以 JEP 243 的形式添加到平台中。这允许你以 Java 接口形式向 JVM 中插入一个 JIT 编译器，其方式类似于将 Java 代理插入到 JVM 中。

该项目的观点是，JIT 编译器其实只需要能够接受 JVM 字节码并生成机器码就可以了。在底层，编译器只是将一个字节 `byte[]`（字节码）转换为另一个 `byte[]`（机器码），所以这可以通过 Java 实现。

这种 Java-in-Java 的方法有很多好处，包括简单和内存安全，以及能够使用标准的 Java 工具链，如 IDE 和调试器，而不必要求编译器开发人员必须掌握深奥难懂的 C++ 技能。

这些优势使 Graal 能够实现强大的新优化，如 10.5.3 节曾提过的部分逃逸分析，这在 C2 中是做不到的。还有一个优势是 Graal 可以让团队为自己的应用程序修改部分内容，比如为定制硬件开发自定义的内部函数，或者开发自己的优化遍[1]。

Truffle 是一个为 JVM 上的编程语言开发解释器的框架。它被设计为一个配合 Graal 工作的库，通过使用一种叫**二寸投影**（futamuru projection）的技术，可以为输入语言从只有解释器自动生成一个高性能的 JIT 编译器。这是一种来自计算机科学学术领域的技术，也称作**部分特化**（partial specialization），最近它在真实系统中的应用更加突出了（不过有些思想在 Python 的 PyPy 实现中已经使用了好几年）。

JVM 上一些现有的语言实现，比如 JRuby、Jython 和 Nashorn，都能在运行时生成字节码。Truffle 提供了一种新的字节码生成方式。到目前为止的性能测试表明，与之前方案相比，Truffle 和 Graal 的组合可能会带来更好的性能。

所有这些工作综合起来就是新的 **Metropolis 项目**。它致力于从 HotSpot 的 JIT 编译器和可能的解释器开始，用 Java 重写更多的虚拟机组件。

Metropolis/Graal 技术已经出现在 Java 9 中，并正式交付，不过它在很大程度上依然是实验

注 1：遍（pass）是编译理论中的一个概念，是对源程序或其等价的中间语言程序从头到尾扫描一次并完成规定任务的过程。——译者注

性的。启用这款新 JIT 编译器的开关是：

```
-XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler
```

Java 9 中还有一个地方会用到 Graal，即预先编译模式。这种模式直接将 Java 编译成机器码，其方式类似于 C 和 C++。Java 9 包含了一个使用 Graal 的 jatoc 命令，该命令的唯一目标是加快启动时间，直到正常的分层编译可以取代它。该工具目前只支持单一平台（Linux/ELF）上的 java.base 模块，但预计在接下来的几个 Java 版本中会支持更多平台。

为了从 Java 类文件中创建二进制文件，可以使用以下新的 jaotc 工具：

```
jaotc --output libHelloWorld.so HelloWorld.class
jaotc --output libjava.base.so --module java.base
```

最后，SubstrateVM 是一个研究项目（也使用了 Graal），它进一步实现了这个功能，并用 Java 编写的整个 JVM 和一个 Java 应用程序一起编译，以生成一个单一的、静态链接的原生可执行文件。这样做的目的是生成不需要安装任何形式的 JVM 的原生二进制文件，而且可以小到几千字节，并在几毫秒内启动。

15.6　字节码的未来方向

虚拟机中最大的变化之一是 invokedynamic 的到来。这个新字节码打开了重新思考如何编写 JVM 字节码的大门。现在尝试拓展在这个操作码中使用的技术，从而为平台提供更大的灵活性。

例如，回顾一下在 9.1 节中讲到的 ldc 和 const 的区别。乍看可能很简单，其实问题复杂得多。接下来看一段简单的代码：

```
public static final String HELLO = "Hello World";
public static final double PI = 3.142;

public void showConstsAndLdc() {
    Object o = null;
    int i = -1;
    i = 0;
    i = 1;
    o = HELLO;
    double d = 0.0;
    d = PI;
}
```

这会生成如下相当直接的字节码序列：

```
public void showConstsAndLdc();
  Code:
     0: aconst_null
     1: astore_1
     2: iconst_m1
     3: istore_2
     4: iconst_0
     5: istore_2
```

```
     6: iconst_1
     7: istore_2
     8: ldc           #3                  // String Hello World
    10: astore_1
    11: dconst_0
    12: dstore_3
    13: ldc2_w        #4                  // double 3.142d
    16: dstore_3
    17: return
```

现在还有一些额外的条目出现在常量池中：

```
    #3 = String             #29           // Hello World
    #4 = Double             3.142d
...
   #29 = Utf8               Hello World
```

基本的模式很清晰："真常量"以 const 指令的形式出现，而从常量池加载的则用 ldc 指令表示。

前者是一小部分有限的常量集，比如基本类型值 0、1 和 null。相比之下，任何不可变的值都可以看作能用 ldc 加载的常量，而且最近的 Java 版本又大大增加了可以放到常量池中的不同常量类型的数目。

例如，考虑一下这段 Java 7（或更高版本）的代码，它利用了 11.7 节中介绍的方法句柄 API：

```
public MethodHandle getToStringMH() throws NoSuchMethodException,
    IllegalAccessException {
    MethodType mt = MethodType.methodType(String.class);
    MethodHandles.Lookup lk = MethodHandles.lookup();
    MethodHandle mh = lk.findVirtual(getClass(), "toString", mt);

    return mh;
}

public void callMH() {
    try {
        MethodHandle mh = getToStringMH();
        Object o = mh.invoke(this, null);
        System.out.println(o);
    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

为了解方法句柄对常量池的影响，下面把这个简单的方法添加到前面 ldc 和 const 的例子中：

```
public void mh() throws Exception {
    MethodType mt = MethodType.methodType(void.class);
    MethodHandle mh = MethodHandles.lookup().findVirtual
    (BytecodePatterns.class, "mh", mt);
}
```

它会生成如下字节码：

```
public void mh() throws java.lang.Exception;
  Code:
     0: getstatic     #6   // Field java/lang/Void.TYPE:Ljava/lang/Class;
     3: invokestatic  #7   // Method java/lang/invoke/MethodType.methodType:
                           // (Ljava/lang/Class;)Ljava/lang/invoke/MethodType;
     6: astore_1
     7: invokestatic  #8   // Method java/lang/invoke/MethodHandles.lookup:
                           // ()Ljava/lang/invoke/MethodHandles$Lookup;
    10: ldc           #2   // class optjava/bc/BytecodePatterns
    12: ldc           #9   // String mh
    14: aload_1
    15: invokevirtual #10  // Method java/lang/invoke/MethodHandles$Lookup.
                           // findVirtual:(Ljava/lang/Class;Ljava/lang/
                           // String;Ljava/lang/invoke/MethodType;)Ljava/
                           // lang/invoke/MethodHandle;
    18: astore_2
    19: return
}
```

可以看到，这里包含了一个额外的 `ldc`，用于加载 `BytecodePatterns.class` 字面常量，它还包含了一个替代用的 `void.class` 对象，用于表示 `java.lang.Void` 类型。不过类常量只是比字符串或基本类型常量**有趣**一点。

这并不是全部，一旦方法句柄发挥作用，将对常量池产生非常大的影响。我们可以看到第一个地方出现了一些新类型的常量池条目：

```
#58 = MethodHandle  #6:#84  // invokestatic java/lang/invoke/LambdaMetafactory.
                            // metafactory:(Ljava/lang/invoke/MethodHandles
                            // $Lookup;Ljava/lang/String;Ljava/lang/invoke/
                            // MethodType;Ljava/lang/invoke/MethodType;Ljava/
                            // lang/invoke/MethodHandle;Ljava/lang/invoke/
                            // MethodType;)Ljava/lang/invoke/CallSite;
#59 = MethodType    #22     // ()V
#60 = MethodHandle  #6:#85  // invokestatic optjava/bc/BytecodePatterns.
                            // lambda$lambda$0:()V
```

这些新类型的常量是支持 `invokedynamic` 所需要的，自 Java 7 以来，平台就在越来越多地使用该技术。整体目标是使通过 `invokedynamic` 调用的方法像典型的 `invokevirtual` 调用一样具有很好的性能，并对 JIT 友好。

未来工作的其他方向包括探索“常量动态化”（constant dynamic）能力的可能性，虽然类似于 `invokedynamic`，但它是用于在链接时无法解析但在第一次遇到时进行计算的常量池条目。

预计在即将到来的 Java 版本中，JVM 的这个领域仍将是一个非常活跃的研究课题。

15.7　并发的未来方向

正如第 2 章中所讨论的，Java 的主要创新之一就是引入了自动内存管理。时至今日，几乎没有开发人员会试图为内存的手动管理辩护，认为这是任何新编程语言都应该使用的积极特性。

我们可以看到，在 Java 实现并发的演进过程中也部分反映了这一点。Java 线程模型的最

初设计是，程序员必须显式管理所有线程，可变状态（mutable state）也必须用锁来保护，而锁本质上是一种协作式设计。如果某一段代码没有正确实现加锁方案，就会破坏对象状态。

> 这可以用 Java 线程的基本原则来表达，即非同步代码既不查看也不关心对象上的锁的状态，可以随意访问或破坏对象的状态。

随着 Java 的发展，后续的版本已经从这种设计转向了更高级别、更少手动操作、通常也更安全的方法——有效地实现**运行时托管的并发**（runtime-managed concurrency）。

最近宣布的 Project Loom 就是该方法的一种尝试。与目前为止在 JVM 上所采用的并发实现方案相比，这个项目希望在更低的层次上支持 JVM 并发。核心 Java 线程的本质问题是每个线程都有一个栈。这样成本很高，而且不能无限制地扩展。一旦我们有了比如说 10 000 个线程，那么用于这些线程的内存就会达到 GB。

还有一种解决方案是后退一步，考虑另一种方法：不能由操作系统直接调度的执行单元，其开销更低，并且可能"大部分时间是空闲的"（也就是不需要执行很高比例的挂钟时间）。

这与其他语言（包括基于 JVM 和非 JVM 的语言）所采取的方法很相似。在很多情况下，它们有更底层的协作式构件，如 goroutine、fiberer 和 continuingation 等。这些抽象必须是协作式而不是抢占式的，因为操作系统看不见它们的协作，而且它们也没有构建自己的可调度实体。

如果要采用这种方法，则需要两个基本组件：被调用代码的表示（比如 `Runnable` 或类似类型）以及调度器组件。具有讽刺意味的是，虽然其他部分还没有到位，但 JVM 从第 7 版开始就有了一个用于这些抽象的不错的调度组件。

Fork/Join API（12.5.3 节介绍过）是在 Java 7 中提供的，因为它基于两个概念：可执行任务的递归分解以及工作窃取（work stealing），从而让空闲线程可以从更繁忙的线程的队列中获取工作。`ForkJoinPool` 执行器是这两个概念的核心，负责实现工作窃取算法。

其实对于大多数任务来说，递归分解并不是非常有用。不过具有工作窃取功能的执行器线程池仍然可以应用于许多不同情况。比如，Akka Actor 框架就采用了 `ForkJoinPool` 作为其执行器。

虽然 Project Loom 还处于早期阶段，但 `ForkJoinPool` 执行器似乎有可能作为这些轻量级执行对象的调度组件。同样，将这种能力标准化到虚拟机和核心库中可以大大减少对外部库的使用需求。

15.8　总结

自第一个版本发布以来，Java 已经发生了很大的变化，即从一开始并没有被明确设计为一种高性能语言，到现在已经成为这样的语言。即使 Java 已扩展到很多新应用领域，但是核心的 Java 平台、社区和生态系统仍然保持着健康和活力。

像 Project Metropolis 和 Graal 等大胆的新举措，正在重新塑造核心虚拟机。`invokedynamic` 也让 Java 走出了其演进的舒适区，为下一个十年重新改造自己。Java 已经表明它不怕进行大胆的更改，比如增加值类型以及重新解决泛型存在的复杂问题。

Java/JVM 性能是一个非常有活力的领域，本章中我们看到了性能在很多领域中仍在取得进步。还有很多其他的项目我们没有时间提及，包括 Java/ 原生代码交互（project panama）和新的垃圾收集器（如 Oracle 的 ZGC）。

因此，本书所讲的内容并不全面，因为性能工程师还有很多东西需要了解。尽管如此，我们还是希望它能够对读者理解 Java 性能世界有所帮助，也为读者的性能之旅提供一些路标。

作者介绍

本杰明·J. 埃文斯（**Benjamin J.Evans**），jClarity 公司联合创始人和技术研究员，Java Champion 荣誉得主，获得过 JavaOne Rockstar 奖，同时还是《Java 程序员修炼之道》一书的合著者之一，经常就 Java 平台、性能、并发及相关主题发表公开演讲。埃文斯还帮助组织了伦敦 Java 社区，并作为社区代表加入了 JCP（Java Community Process）执行委员会，为定义 Java 生态系统的新标准而努力。

詹姆斯·高夫（**James Gough**），Java 开发者和技术图书作者，目前就职于摩根士丹利，专注于构建面向客户的技术。在英国华威大学攻读学位期间，詹姆斯就对 Java 产生了兴趣，毕业后成为伦敦 Java 社区的一员。他曾花 4 年时间在世界上不同国家教授 Java 和 C++，并一直致力于奉献社区，包括参与 JSR-310 的设计和测试，并在 JCP 执行委员会服务多年。

克里斯·纽兰（**Chris Newland**），目前就职于 ADVFN，是一位资深开发人员，使用 Java 处理实时的股市数据。克里斯是 JITWatch 项目（一个开源的日志分析器，能以可视化方式显示和检查 HotSpot JVM 所做的 JIT 编译决策）的作者。他还是 Java Champion 荣誉得主，曾在多个会议上发表过关于 JIT 编译技术的演讲。

封面介绍

本书封面上的动物是一只捻角山羊（capra falconeri）。这种野山羊长着梅林式的胡须和扭曲高耸的角。它们分布在亚洲西部和中部的山区，栖息于海拔 600~3600 米的季风林中。

捻角山羊是草食性动物，主要以各种植物为食，包括草、叶、药草、水果和花。和其他野山羊一样，捻角山羊在其生态系统中发挥着宝贵的作用，因为它们会咀嚼低矮树木的叶子，并通过其粪便帮助植物传播种子。

它们的交配季节发生在冬季，在此期间，雄性山羊通过冲刺、锁角和试图推倒彼此来争斗。随后的分娩会发生在来年的 4 月下旬至 6 月上旬，那时它们会产下一两只幼崽。成年雄性以独居为主，喜欢在森林中生活，而雌性和它们的幼崽则成群结队地生活在高处的岩脊上。

O’Reilly 封面上的许多动物正濒临灭绝，它们是自然界所剩无几的瑰宝。如果你想知道如何保护这些动物，请访问 O’Reilly 网站。

本书封面来自 *Riverside Natural History*。

Java性能优化实践：JVM调优策略、工具与技巧

在当前的互联网开发模式下，系统访问量日增、代码臃肿，各种性能问题纷至沓来。性能优化作为一个常谈常新的话题，受到越来越多开发者的关注。而Java是一门使用广泛的语言，社区生态中积攒了大量宝贵的性能优化经验。

作为一本性能调优方面的实用指南，本书从实验科学的角度将JVM调优的技术原理与方法论相结合，并在此基础上提供了可选择的工具。通过对各方面的深入研究，本书能让使用复杂技术栈的中高级Java技术专家以量化和可验证的方法优化Java应用程序性能。

- **了解Java的原则和技术如何充分利用现代硬件和操作系统**
- **探究一些性能测试以及困扰团队的常见反模式**
- **理解测量Java性能数据的陷阱以及微基准测试的缺点**
- **深入研究JVM垃圾收集日志、监控、调优和工具**
- **探究JIT编译和Java语言性能技术**
- **学习Java集合类API与性能有关的方面，从整体上理解Java并发**

“在过去的20多年里，我花了很多时间来理解JVM的内部细节，在本书中也学到了不少知识。本书文笔流畅，易于阅读，包含了大量对新手程序员和专家都有帮助的信息。不管你是在运行包含200个核心的专用大型机器，还是运行资源受限的Linux容器，本书都可以帮你挖掘出Java应用程序的巨大性能。”

——Christine H. Flood
红帽公司

本杰明·J. 埃文斯（Benjamin J. Evans），初创公司jClarity联合创始人和技术Fellow，获得过Java Champion荣誉和JavaOne Rockstar奖，《Java程序员修炼之道》合著者。

詹姆斯·高夫（James Gough），Java开发者和技术图书作者，任职于摩根士丹利，专注构建面向客户的技术。

克里斯·纽兰（Chris Newland），JITWatch项目作者，Java Champion荣誉得主，ADVFN高级开发者和团队负责人，擅长使用Java来处理实时的股市数据。

封面设计：Randy Comer　张健

图灵社区：iTuring.cn

分类建议　计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

扫码领取
随书代码资料

ISBN 978-7-115-55233-4

定价：109.00元